

AMOS

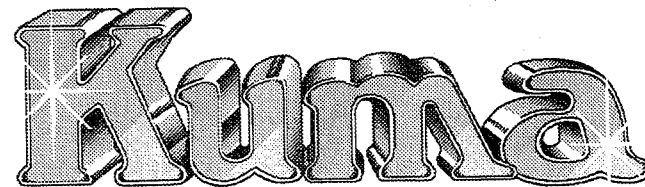
In Action

Anne & Len Tucker

AMOS IN ACTION

by Anne & Len Tucker

ISBN 07457 0221 X

The word "Kuma" is rendered in a highly stylized, three-dimensional font. The letters are thick and blocky, with a metallic or chrome-like finish indicated by highlights and shadows. The 'K' is particularly large and prominent, with a bright starburst effect on its left side. The 'u' and 'm' are also large and rounded, while the 'a' is smaller and more compact. The overall style is reminiscent of classic comic book lettering or retro sci-fi typography.

AMOS IN ACTION

©1992 Anne & Len Tucker

This book and the programs within are supplied in the belief that the contents are correct and that they operate as specified, but the authors and Kuma Computers Ltd shall not be liable in any circumstances whatsoever for any direct or indirect loss or damage to property incurred or suffered by the customer or any other person as a result of any fault or defect in the information contained herein.

ALL RIGHTS RESERVED

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, scanning, recording or otherwise without the prior written permission of the author and the publisher.

Published by:

Kuma Computers Ltd
12 Horseshoe Park
Pangbourne
Berks
RG8 7JW

Tel 0734 844335
Fax 0734 844339

AMOS IN ACTION

C O N T E N T S

CHAPTER ONE THE AMOS FAMILY	1
CHAPTER TWO HOW IT ALL BEGAN	7
CHAPTER THREE DESIGNING AMOS GAMES	17
CHAPTER FOUR INSIDE TOME	23
CHAPTER FIVE MUSIC AND SOUND EFFECTS	27
CHAPTER SIX STRUCTURING YOUR PLANS	29
CHAPTER SEVEN BRIEF TOUR OF MARVIN	31
CHAPTER EIGHT MARVIN'S PROCEDURES	39
CHAPTER NINE THE AMOS COMMANDS IN MARVIN	95
CHAPTER TEN USING PROCEDURES IN YOUR PROGRAMS	105
CHAPTER ELEVEN AMOS HINTS AND TIPS	109
CHAPTER TWELVE AMOS UTILITIES	115
APPENDIX ONE ABOUT THE AUTHORS	129
APPENDIX TWO A LIST OF EVERY AMOS CLUB WORLDWIDE	133
APPENDIX THREE HOW TO GET YOUR FREE GAME	135
APPENDIX FOUR SPECIAL OFFERS TO ALL AMOS USERS	

Other Computing Titles From Kuma:

Commodore Amiga

Program Design Techniques for the Amiga by Paul Overaa 07457 0032 2
Intuition A Practical Programmers Guide by Mike Nelson 07457 0143 4

The Little Red Workbench 1.3 Book by Mark Smiddy 07457 0048 9
The Little Blue Workbench 2 Book by Mark Smiddy 07457 0055 1
Amiga Five-0 The Top 50 Amiga Games by Ashley Cotter-Cairns 07457 0148 5

IBM PC & Compatible Micros

DTP Sourcebook - Fonts & Clip Art for the PC by J. L. Borman 07457 0030 6
PageMaker 4.0 for Windows by William B. Sanders 07457 0031 4
The Windows Guide Book by Gill Gerhardi, Vic Gerhardi & Andy Berry 07457 0041 1
A Practical Guide to Timeworks Publisher 2 on the PC by Terry Freedman 07457 0147 7

The DR DOS 6 Quick Start Guide by John Sumner 07457 0038 1
Illustrated DR DOS 6 - The First 20 Hours by John Sumner 07457 0044 6
The User's Guide to Money Manager PC by John Sumner 07457 0047 0

Breaking Into Windows 3.1 by Bill Stott & Mark Brearley 07457 0056 X
The Utter Novice Guide to Q Basic by Bill Aitken 07457 0046 2
PagePlus Illustrated by Richard Hunt 07457 0062 4
DOS 5 Quick Start Guide by John Sumner 07457 0054 3

Psion Organiser

Psion Organiser Deciphered by Gill Gerhardi, Vic Gerhardi & Andy Berry 07457 0139 6
Using & Programming the Psion Organiser by Mike Shaw 07457 0134 5
File Handling on the Psion Organiser by Mike Shaw 07457 0135 3
Machine Code Programming on the Psion Organiser 2nd Ed. by Bill Aitken 07457 0138 8
Psion Organiser Comms Handbook by Gill & Vic Gerhardi & Andy Berry 07457 0154 X

Psion Series 3

First Steps in Programming the Psion Series 3 by Mike Shaw 07457 0145 0
Serious Programming on the Psion Series 3 by Bill Aitken 07457 0035 7
Introduction to Using the Psion series 3 by Rod Lawton 07457 0146 9

Cambridge Z88

Z88 Magic by Gill Gerhardi, Vic Gerhardi & Andy Berry 07457 0137 X

Games

Sega Megadrive Secrets by Rusel deMaria 07457 0037 3
Sega Megadrive Secrets Volume 2 by Rusel deMaria 07457 0043 8

Corish's Computer Games Guide by Dean Corish 07457 0150 7
Awesome Sega Megadrive Secrets by J. Douglas Arnold 07457 0226 0
Awesome sega Secrets II by J. Douglas Arnold 07457 0214 7

Corish's Console Games Guide by Dean Corish 07457 0084 5
Awesome Nintendo Secrets by J. Douglas Arnold 07457 0216 3

Sharp IQ 7000 & 8000

Using Basic on the Sharp IQ by John Sumner 07457 0034 9

A selection from our fast-expanding range - latest full details on request

Amiga Books From Kuma

Intuition - A Practical Amiga Programmers Guide

BY MIKE NELSON

ISBN number 07457 0143 4 Cost £16.95

There is no reason why anyone with experience in Basic can not graduate to the world of C and Intuition, the Amiga's Graphical User Interface. "Practical" is the keyword describing this highly readable book. Extensive examples are provided throughout the text, it being the author's premise that the best way to start learning to program is looking at and modifying other people's code. The programs can be typed in, or obtained on disk separately.

Highly detailed explanations of the code given in the book are given. All code is presented in C, but the information will serve anyone programming in other languages, especially Assembler.

An essential book for all Amiga programmers.

Mark Smiddy's Little Blue Workbench 2 Book

ISBN number 07457 0055 1 Cost £14.95

Conceived and written by one of the top Amiga writers, this book will help Amiga owners get the best from this exciting machine. Mark Smiddy is well known in the Amiga world for his ability to impart his very considerable expertise in a polished, easy to read style which will benefit users of all levels.

The book is packed with tutorials covering all major workbench functions these are supported by numerous practical tips and examples. The most common source of problems for users are printers and printing, these areas are given extensive coverage in the books which also include tables of printers and compatible drivers. Also included is a hard and software buyers guide, a guide on what to do when things go wrong, an extensive glossary to explain "computerspeke" and a detailed index.

An invaluable guides for all Amiga users.

Amiga Five-0 - The Top 50 Amiga Games of All Time

by Ashley Cotter-Cairns

Price £9.95

ISBN 07457 0148 5

With some classic games appearing at bargain prices and a glut of new titles arriving, the choice of games can be confusing and many players get frustratingly stuck. Ashley Cotter-Cairns, a leading expert on Amiga games a well known name in Amiga games playing journalism, gives the answers.

In depth playing guides, tips and mini reviews on 50 of the best ever games for the Amiga including: Also included is advice on controllers, other games of similar types and what to add to your Amiga to make it into a killer games machine.

Project X

Rainbow Islands

Supercars 2

Zool

Chessmaster 2100

Populous

Sim City

Utopia

Beast III

Moonstone

Jimmy White Whirlwind Snooker

Kick Off 2

Lotus Turbo Challenge 2

Lotus III The Ultimate Challenge

Sensible Soccer

Gobliins

Lemmings & Oh No More Lemmings!

Super Tetris

King's Quest V

Legend

Legend of Kyrandia

Might and Magic III

Air Support

Elite

Crucial information for all committed Amiga games players.

CHAPTER ONE

THE AMOS FAMILY

To start off this journey into Amos programming, we would like to introduce you to the people who will be able to help you get the most from this marvellous programming language and also other programs and hardware that will be useful to you whether you are an enthusiastic learner or someone who wants to write commercial quality games.

AARON FOTHERGILL

Author of Amos TOME, owner of SHADOW SOFTWARE, and head of The AMOS CLUB.

The Amos Club is run by Aaron Fothergill for all Amos users. For your annual subscription, you get six issues of the club newsletter, access to Aaron's Helpline number and discount on Shadow Software's TOME extension.

Aaron has also produced several games and two utilities, which are essential for the serious Amos programmer. These are SPRITEX, a powerful bob and sprite editor, which we used extensively during the development of Marvin the Martian, and CTEXT which unfortunately was not released until Marvin was completed. All of these are available from Deja Vu Software (see below for address and further information)

For more details of the Club, TOME, SPRITEX or CTEXT send SAE to:-

The Amos Club, 1 Lower Moor, Whiddon Valley, Barnstaple, North Devon. EX32 8NW

SANDRA SHARKEY.

Proprietor of Deja Vu Software who sell an extensive range of Licenseware software.

Sandra's Library also contains Deja Vu Professional Licensed software, previously known as Licenseware. These disks cost a bit more than the PD disks as the price includes a royalty fee which is paid to the programmers. Marvin The Martian is just one title from a range of nearly 70 titles presently available.

To get more details and information of the Deja† Vu Software Club, send SAE to :-

Deja† Vu Software, c/o Sandra Sharkey, 7 Hollingbrook, Beach Hill, Wigan. WN6 7SG

LEN & ANNE TUCKER.

We can also offer you help with Amos. Apart from writing this book we produce a disk magazine called "Totally Amos". This is published every two months starting November 1991, and is available as single issues or as a yearly subscription.

We are trying to provide a place where all Amos enthusiasts are welcome to share their ideas, programs and problems.

There is a special section for children with a lesson wrapped around a little program written specially for them. We would like to show the adult users of Amos that it is a good idea to let the kids have a go at programming with Amos.

We have reviews on Amos produced programs whether they are Public Domain, Licenseware or Commercial releases.

To help you make your programs more professional, there are hints and tips and routines for you to use in your own work.

We have now just taken over the official AMOS club from Deja Vu. To receive details on this club write to us at the address below.

For details send SAE to:-

"TOTALLY AMOS", 1 Penmynydd Road, Penlan, Swansea. SA5 7EH

Over the past few months when we have been producing Totally Amos, we have received great help and support from Paul Townsend, of Technical Fred Software, who is based in the Manchester area. Paul has contributed lots of handy programs to the magazine and is willing to offer his help to other Amos users. He is available for a chat on the phone after 7pm most evenings and will also be able to help with Cli and Workbench problems. If you're greeted by Paul's answerphone, leave your name and tell him the nature of your problem, and phone him back later.

Contact Paul on 061 7037842 after 7pm when he'll be pleased to talk to you. This is also a Fax number.

THE AMOS UTILITIES

Now you have been introduced to the people, let's get on to tell you about other utilities we use which help to make better programs.

The most obvious program is AMOS itself. Over the last eighteen months we have found

out just how versatile a language this is. With AMOS, just about any program is possible, your imagination is the limit for ideas.

The **AMOS COMPILER** is another must if you want software houses to take your work seriously. Many will not look at anything which is written in basic and uncompiled, although to be fair some will. Programs run differently when compiled, the action is faster and smoother thus giving better gameplay. There are two versions of Marvin The Martian on the disk given with this book. One is compiled, one is source code, that means that you have to load it inside AMOS. Play the compiled version, then if you have AMOS, load the source version into AMOS, run it and see the difference for yourself!

Another advantage of a compiled program is that your program code is protected from prying eyes! Remember to keep an uncompiled copy of anything that you compile, as you will not be able to get into the listing again if you don't.

AMOS 3-D is an extension to Amos that allows you to create 3-D graphics and manipulate them inside AMOS. There is an object modeler which allows the creation of the objects and also new commands to use to attach them to your program.

All three products are available from:- Europress Software, Europa House, Addlington Park, Macclesfield, Cheshire. SK10 4NP.

AMOS costs £49.99, the Compiler (needs AMOS) £29.99, and AMOS 3-D (also needing AMOS) £34.99.

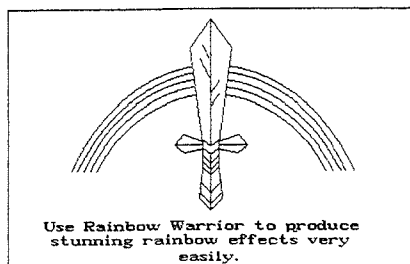
TOME, this is the T^Otal Map Editor written by Aaron Fothergill for use with AMOS. It is the main driving force behind Marvin the Martian. With TOME you can design your game world and scroll around it very easily. It also provides several shortcuts for programming and is a must for the enthusiastic programmer. Tome, like the Compiler, is an AMOS extension.

TOME is available from Shadow Software at the address given earlier.

No program is complete without graphics, so an Art Package is an essential part of your equipment. Most Amigas are sold with one, mostly Deluxe Paint from Electronic Arts, and this is the one we'll be referring to in the book. Any version is suitable, but if you can, go for the newest version that you can afford.

Music can come a lot cheaper! We have been using SoundTracker which is a Public Domain program obtainable from most PD Libraries. This needs instrument disks which are available on PD in abundance and can be collected over time.

Another similar program is the Games Music Creator (GMC) also available from PD libraries. Amos accepts converted files from both these programs.



If you've seen the high score screen in the game, you'll see that the names are printed on rainbow bars. These are created with a Public Domain Program called RAINBOW WARRIOR, written by Martyn Brown, also known as 'Spadge'. These effects could be created manually, but it would take a lot of time and knowledge, Rainbow Warrior saves out the necessary data which you then merge into your Amos program.

Rainbow Warrior is available from Deja Vu Software, address as given earlier, disk number APD 76.

Another couple of handy programs are also available from Deja Vu. These are both Licenseware and are written by Aaron Fothergill.

The first is Sprite-X which is a sprite editor used for grabbing sprites, cutting them off an IFF screen and storing them as a sprite/bob bank. It's very useful and is far better than the basic version supplied with Amos.

C-Text is another Licenseware disk programmed by Aaron Fothergill. It is an amazing little extension that allows you to use multi-coloured fonts as text in your programs, which allows greater flexibility than the Get Disc Fonts command. These disks cost £3.50 each from Deja Vu, and are disks LPD 55 (Sprite-X) and LPD 56 (C-Text).

When you start to program seriously, you will find that a good file copier is essential as you will often need to move files around to meet your needs. Diskmaster 1.3 is highly recommended as a commercially released program, but there is also a Public Domain file copier called Sid, which will do the same as Diskmaster, but costs less!

Some examples of hardware that will be useful to you are the following:-

A MINIMUM of 1 megabyte of memory. You will not get very far with just 512k as AMOS needs a lot of memory to work in.

An upgrade to 1 meg costs less than the price of a decent game and is the one ESSENTIAL hardware purchase.

A second disk drive is also on this list, but is not as essential as the extra memory. Disk swapping can be a bind, and after owning a second drive for a while, you'll wonder how you managed without one!

A monitor is another item to be considered if you are using a portable T.V. with your Amiga. It is by no means an essential to programming, but it does make text easier to read, and the graphics a lot clearer.

The REAL luxury must be a hard drive. You can store all the programs you need on it and load each one at the speed of light as it is needed eg when programming using Amos, you may need to pop into DPaint to alter a graphic, move to Diskmaster to copy over a file from a floppy disk and then go back to Amos. That journey takes quite a while to load from floppy disks, but each program loads in no time from a hard drive.

There are often letters in the popular computer magazines complaining about 'having to buy' so many extra things to program with Amos. We would like to think that this book will go some way towards dispelling this idea.

Yes, Amos itself has a recommended price of £49.99, but as with most things, shopping around will find you discount prices.

The Compiler is a must for serious users, but is not necessary to start off with, similar can be said for Amos 3-D. It's a must to write 3-D games, but not for starting to learn.

One letter stated that there were too many updates. This was a problem when Amos first appeared, but now things are running smoother, updating is less frequent. It has always been the policy to supply updates on PD so that they can be obtained easily and cheaply. Some other companies would have expected customers to buy the latest versions at the full price.

Another complaint was that there were hundreds of Amos PD disks to buy! Of course, you do not have to buy them all, or any if you do not want to, but there are hundreds you can choose from if you want to see what other people are doing with Amos. The extras you will need can be bought gradually as you feel they are necessary, you should never feel pressured into buying things before you need them.

We first heard about Amos about a year before it was released having seen STOS in a shop display. We knew it was just what we were looking for, although we didn't know at the time that the Amiga version would not be available for so long after this. While we were waiting, we upgraded our 1/2 meg machine to 1 meg, and bought a second disk

Chapter 1

drive. This was enough to get us going and it was almost a year before we needed a further 1 meg of memory.

This shows that upgrading your system need not be done overnight, but it can be built up if and when necessary. Remember that YOU are an essential part of programming, the biggest and best computer with all the available add ons will not produce a program. With the minimum of a 1 meg Amiga, and a copy of Amos you can make things happen!

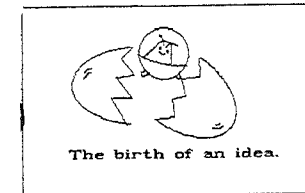
CHAPTER 2

HOW IT ALL BEGAN

It can be very frustrating trying to find a game that you really will enjoy playing. You must have read reviews and advertisements for games and have thought that at last you've found the game that was meant to be played and enjoyed by you.

Unfortunately, once you have saved up enough to buy your dream game and have got home and loaded it into your Amiga, you find that the advert was misleading and that the reviewer must have been looking at another game when he wrote the article, this isn't your type of game at all!

So, what can you do about it? The disk functions perfectly, so you can't take it back, so you save up again to buy another game which you hope will be more to your liking.



Or, you can write your own game!

"I can't do that!" you say, - We say "you can!"

With a copy of AMOS, which you can now buy for the price of a game, and this book, which costs a bit less than a game, you CAN write your own game. Or at least be well on your way to doing so.

This is how Marvin The Martian came to life in the mind of our 13 year old daughter.

The very first we heard of Marvin The Martian, was when Melanie told us that she was designing a 'comic on a disk' with a martian as the main character. She hadn't got very far when it occurred to her that Marvin's story could be put into a game.

Melanie has never really found a game that she could thoroughly enjoy, but she has always liked maze type games that have objects to collect, doors that need keys to get through them and beasts to avoid. There are a couple that have these features, but they don't have the gameplay she'd like to go with them.

The story of Marvin seemed like a fair idea for this type of game, so she was encouraged to expand on the idea with a view to us writing a maze game for her. Then we came up with a better idea, why shouldn't she help out with the graphics and the coding? (You're never too young (or too old) to use AMOS!).

So the epic journey of Marvin had begun. It was an epic because it took ten months to get released, due to various problems during its production. We'll tell you how you can avoid the same pitfalls during this book. As the saying goes, never say die, there's usually a way out of any problem that arises!

If we were to write Marvin again, the coding would be a lot different. If you program on a regular basis, you will be constantly learning new things, such as shortcuts to achieve the same effect, faster routines for movement etc. No one can say that they know all there is to know about programming as new things are being found out all the time. This is why we feel that it is important that everyone shares what they know, in this way lots of people get to know the tips that will help them to get the best from a system like AMOS.

If you are willing to help other people, in time, other people will help you and your knowledge will increase. Since Marvin was written we have learnt things that might have made the program faster, smaller and possibly more structured.

You could rewrite your program every time you learn something new, but this isn't really practical as then the program would never get finished!

Work on the theory that the program will be OK if it works and you can understand it enough to know which part to look at if a bug is found ie if there is a problem in Marvin with the enemies movements, we know to look in the Procedure OTHERBLOKE.

There could be people who criticise the way you code, ie "What a messy

program!" or "Not very structured, is it?" This may be the case, ours is usually one or the other in other people's views, but we can not only read them, but also understand them. So as far as we are concerned, they are structured - to our needs anyway!

Every coder has his or her own way of writing and structuring a program, and also there are many different ways that you could structure a program, so go for the one that suits you the best. When it comes to the crunch, the only person that you have to consider pleasing when you are developing is yourself. The users of your program will only be interested in what they see when the program is run, and that it functions correctly.

So, off we go!

THE BEST LAID PLANS....

The first thing to do is to write the game specifications.

Don't panic! All this means is that you decide what you want the game to do, how you want it to happen, what bonuses will be available, what beastsies will kill you etc etc.

Let's look at our game specs for Marvin (Programs should always be given a "pet" name, they don't like being called IT, and you will avoid confusion when trying to tell your friends about your program!)

There are advantages to writing specs, you at least know what you're going to be writing, and you will be able to plan out your work to cover all the aspects of your game. It does help to know where your going, even if you're not too sure how to get there.

Don't take these plans as hard and fast rules, the best ideas seem to come up as the program is being written with bits getting changed around to make the gameplay more interesting.

TYPE OF GAME: Marvin is to be a maze game that allows you to move over an area bigger than the viewing area of the screen.

AIM OF GAME: Marvin the Martian has to get through a series of mazes in order to rescue his girlfriend Meryl who has been kidnapped by a monster.

CONTROL OF GAME: Joystick to move Marvin around the mazes, mouse and keyboard to input name for hi-score, and to choose options at start of game.

FRIEND: What is your main character to be? Marvin is a martian, therefore he could be in a spaceship which flies around the map.

ENEMIES: Characters will appear at random. They will deplete your energy if you bump in to them, they can be eliminated, but they should not be shot. Drop an oil slick and they will die if they slip on it!

BONUS FEATURES: Sometimes a beast will leave a coin when he dies, these will give you extra lives if you have enough money. There will be other objects to pick up which will give extra points, but they are not essential.

WHAT TASKS ARE TO BE COMPLETED AS YOU MOVE AROUND: There will be a display showing you which objects have to be collected before you get to the next level. The numbers next to them show you how many are left to get.

ENERGY FACTOR: Energy level is shown as two bars on the control panel. If you bump into a beastly, your energy goes down quite a bit. You also lose energy as you move around. If your energy runs out completely, you lose a life, lose three lives and its GAME OVER!!

KEYS AND DOORS: There are four differently coloured keys scattered around the maze. They can each be used once to open a door of the same colour. Things you must get will be behind these doors, there should be more keys than doors, so it will be possible to complete the game.

OTHER BITS AND PIECES: Design a title screen that gives an idea of the game, also decide how the credits are to be displayed. Options such as 'start game' 'help' 'hi scores' should be given here. Design a screen to display information about the game. Design a hi-score table and decide the score needed to get onto this table. Also, design a way of putting a name into the high score table.

The game is to have fifteen levels, split into three sections of five levels each. Each section will be bigger than the previous one.

As it will be impossible to complete the game in one session, there should be a way of getting back to the level you were on - giving a password at the end of each level will allow the player to re-enter the game, remember to design a screen where this password can be entered.

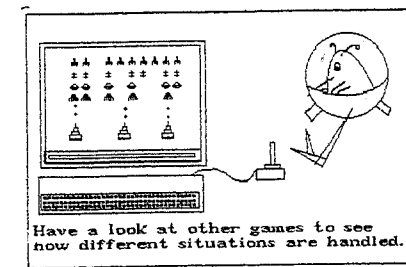
There should be an end screen, showing Marvin And Meryl going home.

Remember that the notes you make now are not legally binding, if you get better ideas as you program, use them! One exception to this is if you are working from someone else's specifications, then you should discuss with them any changes you wish to make.

After the game is finished, go back to your original notes and see how much they've changed!

RESEARCHING YOUR SUBJECT

You should now have a good idea of how you want your game to look and play. Look at other games to find the problems you could be faced with. Look at the way the characters interact with each other. What has to be done to get rid of an enemy, do you bump into it, shoot it once, shoot it repeatedly etc. Watch out for special effects, how do the screens change? Do they fade out, use screen wipes, or can you find other ways of changing the picture.



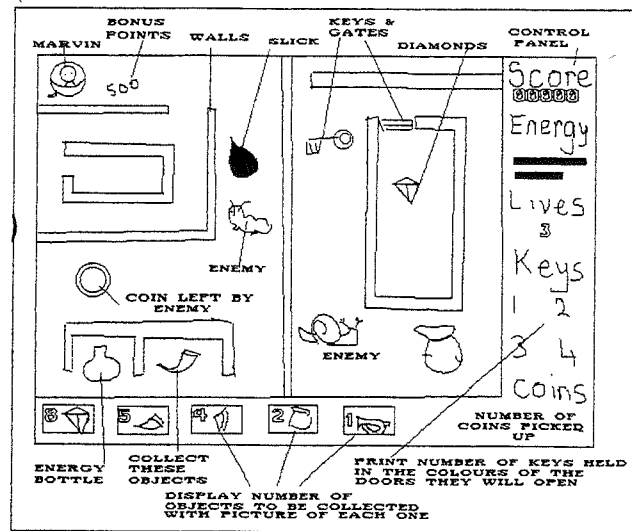
Another source of ideas are the disks in the Amos Public Domain Library. There are plenty of demos and source code disks where you will find special

effects done in Amos.

They should not be used in any commercial program without the Author's permission or at the least if he/she cannot be contacted, credit should be given for anything you use.

Drawing a 'thumbnail' sketch of how you want the game to look on the screen will bring the game to life. This is very useful as a guide to drawing your display and is essential if someone else is drawing the graphics for you. It is only necessary to do a rough sketch, please look at the diagram of Marvin's sketch as an example.

Using these guidelines, you should now be able to start your game.



IN THE BEGINNING..

First of all look at what you are going to ask the program to do. We wanted our character, Marvin, to move through a series of mazes. Originally, before the above set of specs were written, the mazes were going to be single screens, and there were going to be over a hundred of them. These would have to be individually designed and processed before they could be used in

the game, work was started, but this proved to be too time costly, so Marvin was put away for a while.

A month or so later, Marvin was brought out again, this time we decided to write our own driver program to move a tiled map around, but this idea was also discarded as being too memory hungry and the movement was too jerky for good game play.

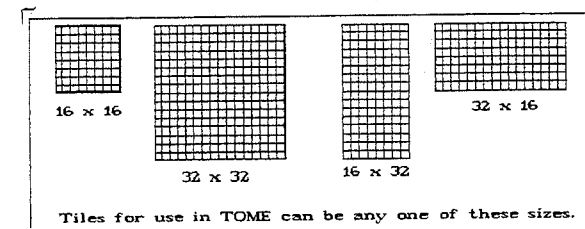
A couple of months later, a new Extension was released for AMOS. TOME was written by Aaron Fothergill and released by Shadow Software, and was just what we needed to get Marvin moving.

TOME stands for Total Map Editor, with this amazingly easy to use program, it was now possible to write a maze game which allows eight way scrolling around an ENORMOUS map.

This is when we changed the original idea to the larger but fewer level plan. This idea gave far better scope for gameplay, and it turned what had been a possible kiddies' PD game into a contender for Licenseware, where it could earn some royalty payments.

So, we decided to use TOME to put the maps of the mazes into the game. The next step was to decide what sort of graphics to use and how to go about getting them.

TOME has certain rules that have to be obeyed in order to make the program work. TOME uses tiles or rectangles which fit together to form the whole map, just like a jigsaw puzzle. There are four sizes of tile that can be used, these are:-



This therefore governed the size of the character sprites to be used in the

game.

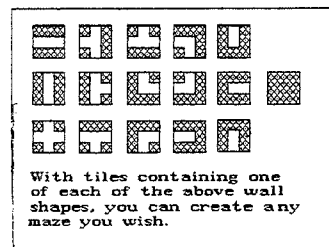
We chose 16 x 16 tiles as this gave the most flexibility for creating the mazes so we had to be sure that the sprites would fit into the spaces between the walls of the mazes. It would not look very professional to have your characters balancing half on the walls and half on the floor of the maze.

PLANNING YOUR GRAPHICS

To make sure that you prepare all the graphics you will need, it is a good idea to make a 'shopping list' of the things to be drawn.

The list for Marvin went something like this, you'll always leave something off this list, but it a good starting point.

MAZES: Tiles for the walls, these should contain all the elements needed to construct a maze, don't forget that you'll have to include the corners, T-joints as well as the straight bits. see diag)



As well as the walls of the maze, you will need a floor tile so that your maze isn't suspended in space - unless of course you want it to be!

CHARACTERS: There is one obvious character, and that is Marvin. We shall refer to him as the 'friend' to avoid confusion. He is the one constant character in the game.

You will also need 'enemies' you must decide how many of these will be used.

Some of the sprites used in Marvin the Martian can be found on the Sprites

600 disk which comes with AMOS. The Marvin character is from this disk as are some of the enemies found in the earlier levels of the game. (Used here with kind permission of Europress Software)

OBJECTS: Next you must have the objects that are to be collected as Marvin goes through the mazes.

PANEL: To give the player information on his/her progress, you will need to have a control or information panel.

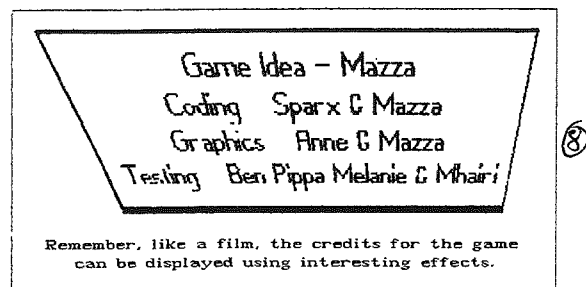
Write down what you want to display on it before you start drawing, as it is often hard to change the layout of a panel to include a new idea later on.

TITLE SCREEN: The title screen is the first thing that your player will see, so this should give a representation of the game. Marvin's title screen shows Marvin running from one of the enemies, with the walls of the maze in the background. The screen also gives the options for the game - HELP HI-SCORES PASSWORD LEVEL and START GAME.

END SCREEN: What happens at the end of the game? Do you just have GAME OVER or THE END printed on a blank screen, or will the player be rewarded with a specially designed picture and maybe a spot of animation? We decided that a reward for completing all the levels was needed, so we added an end screen to the graphics shopping list.

DATA INPUT DEVICES: We also needed a way that the player could input his/her name on reaching the high score table and a way of entering a password. We thought that there should be two control gadgets, rather like calculators where this could be done, one with arrows and an enter key for letters, and the other with number buttons, for the password (or passnumber, in this case)

CREDITS: You need to think of how you will display the game credits, but as you may not know all the people to credit before you start, just remember that it will need graphics.



That should cover most of the preparation for the game, however dreary it seems, it is necessary to have a plan of what you are going to do. It will save you a great deal of time later on, and help in the structuring of the program.

CHAPTER THREE

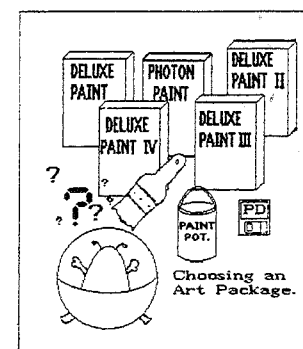
DESIGNING AMOS GAMES

Contrary to most people's belief, the next step in writing a game is to load an art package, NOT Amos! If you stop for a minute to think, how far will you get with programming, if all you've got to work with is a blank screen?

Not everyone can draw, and not everyone can code, we are lucky that when we write as SPARX we can do both between us. If you're a coder but cannot draw, don't give up as there are plenty of graphics on PD which you can use to get you started. Even the crudest of graphics can be used to begin with, pin men and boxes are enough for what is known as the 'story board' of a potential game. Once the game is well under way, and you have more confidence, you can start looking for an artist to produce the graphics for you. Since we started producing 'Totally Amos' we have found that there are plenty of artists who need coders, coders who need artists and musicians who need both. This is one way of finding them.

THE TOOLS OF THE TRADE

Our favourite art package is Deluxe Paint from Electronic Arts, and as this is given with just about every Amiga sold, most people own it. This is the program we shall refer to in this book.



A decent mouse will help produce decent graphics, a badly functioning

mouse that needs a heavy hand to get any sort of response will cause nothing but frustration. It is well worth investing in a new mouse if yours is sick. You can now get excellent upgrades for under £25, less than the price of some games!

The best drawing tool has to be a graphics tablet, these have come down to a more reasonable price and are worth investing in if you do a lot of graphic work. Remember though, they cannot make you into a great artist, but they do make things easier if you are more used to drawing with a pencil.

The version of DPaint you use is irrelevant, Version 1 is good enough, but the later versions give you more effects to play with. Dpaint 3 has animation, which is very handy, DPaint 4 has a great deal more and must be the ultimate Art Package. It does a lot of the hard work for you and can produce stunning effects at the click of a mouse button!

LOAD YOUR WEAPON

Having loaded your Art Package, the next thing to decide is the number of colours that you want to use, and how to arrange them in the palette.

The main aim when you're programming should be to use as little memory as possible, so bear this in mind when you choose your palette. Sixteen colours will use much less memory than thirty two, in low res, and will help your game to run faster.

Having said that, Marvin has 32 colours, this is mainly because it was all planned well before we realised the benefits of fewer colour. We tried to convert the graphics to 16 colours, but it would have taken a lot of work to get the same effects again, so he stayed as he was.

ON THE TILES!

The best thing to start with is the background as then you will be able to set the scene.

As we are using TOME, the first graphics will be the floor and wall tiles that make up the map of Level 1. It was decided that the floors for each section would stay the same, with the walls being different for each level. The

designs did not necessarily have to be different, a change of colour will often be enough to give a different feel to a level. If each set of five levels had the same set of wall tiles, the game would be boring to look at, and the player would not feel as if he/she was achieving anything new by getting onto a higher level. If you opt for a simple change of colours in different levels, you could save time and memory by using colours from a certain part of the palette and then changing the colour numbers inside Amos instead of drawing the tiles in each set of colours.

Our tiles are 16 x 16 pixels, all background graphics have to conform to this, as the tiles will be placed onto a grid like a geometric jigsaw puzzle.

To help you to stick to these rules, draw an outline square which is 18 x 18 pixels, to act as a frame, you will then be able to draw inside this frame and know that you will not breach the tile size. (Likewise if you are using another tile size, draw a frame that is two pixels bigger than the size of the tile and draw inside this frame.) Remember though, to grab the area **INSIDE** the box when storing the tiles for use in TOME.

There should be one tile that only has the floor design on it, the wall tiles in Marvin have the actual wall piece and also a few pixels either side of the floor design which gives the illusion of wider paths between the walls. Be sure that the tile edges match, or the joins between the tiles will be obvious and give a chequered effect.

The objects that Marvin has to pick up must also be on tiles, the tiles are switched for the plain floor tiles to make it look as if the object has been picked up.

Again, use your frame to gauge the size of your objects. Draw them on a plain background, grab them one at a time as a brush, then stamp them onto a copy of the floor tile.

The same goes for the doors that Marvin opens, remember that the walls run horizontally **AND** vertically, so each door will have to be drawn showing two views. Marvin drops oil slicks to get rid of the enemies, and these are tiles as are the numbers displayed under an object when it is picked up.

The way to store these tiles for use with TOME will be explained a bit later on.

GOODIES AND BADDIES

The graphics for your friend and his enemies must fit into the same frame as the tiles so that it looks as if they are moving inside the maze and not hovering above the walls. This is a rule needed for a game like this, but it is not an essential rule for all TOME games, your sprites can be any size. Another point to make is that Marvin's sprite could have been smaller than 16 x 16, but detecting collisions with other objects would have required more programming.

As these graphics move, they should be animated, the enemies need between three and five frames to move about. They do not need to be complex, simple changes between frames should be enough to make them look as if they are walking or crawling rather than sliding.

The friend character needs more thought as he is the focus of attention and has more functions to perform. He must move properly and if he bumps into an enemy, should explode. This takes quite a few frames, but the effect is worthwhile. The fact that Marvin is inside a ship made animation easier as a realistic walking action did not have to be taken into account.

MAIN CONTROL PANEL

In order to keep your player informed of how he/she is progressing, you need a control panel. In Marvin, most of the information is displayed in a panel on the left of the screen with the number of items to be collected shown in a strip along the bottom of the screen.

The main panel displays the score, energy, lives left, the number of keys found, and the amount of coins won while the bottom strips shows pictures and the number of the objects to be collected.

Play around with fonts to find one that suits the style of the game, an LED display type font would look totally out of place in a game set in the Middle Ages and so on. Make sure that the text fits inside the allotted space of the panel, try to make the text a bit different from the original by outlining it, as in Marvin or shadowing with a darker shade.

Remember that you will need to draw everything you need for the game. For

a game that has a high score table and password you will need some way for the player to input the information. We decided on screens showing remote control-like devices.

Now all that's needed is a title screen and an end of game reward screen.

You won't need all of the graphics before you start coding, but the TOME tiles and your main character should be the minimum.

CHAPTER FOUR

INSIDE TOME

As TOME played such a large part in the writing of MARVIN THE MARTIAN, it's only right that we spend a little time telling you how it works so that you can follow the programming.

TOME is an extension to AMOS which means that you add it to your copy of AMOS as described in the instructions, so that you have over thirty new commands at your disposal to use in your programs. There is a map editor supplied with AMOS, but this is very basic in comparison with TOME which is bursting with tools and gadgets to help you create your world. With the latest version you can view a miniature display of your map at the press of a key. This would have been invaluable to us if it had been in the original version, and will be made full use of in future games!

As described earlier, you draw your tiles in Deluxe Paint, or other Art Package. You store them on a screen as follows. Grab each tile as a brush and place on a spare screen. They MUST be stored touching, but not overlapping each other in horizontal rows across the screen starting at co-ordinates 0,0 on the art package screen. This is vital as they will be taken into TOME starting at this point and even one pixel out will give weird results.

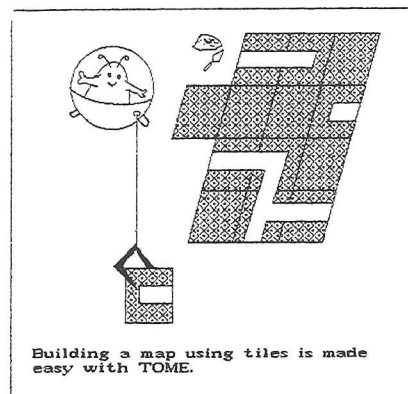
You then load them into the tile cutter supplied with TOME. This cuts the screen up into individual tiles and if you wish will discard any blank or duplicated tiles, which saves memory.

Creating a world in TOME is child's play it's just a matter of clicking on the tile you want and placing it where you want it to be, jigsaw fashion.

You can assign values to tiles, each tile can have a maximum of four values assigned to it in four separate lists numbered 0 - 3.

As an example, we have a tile showing a sack of gold. Let's say that it is on tile no 26, tile val list 0, the tile value for that tile on your first list corresponds to tile 18 which shows '5000 points' as a score for getting the sack, then the tile value for the original list is tile 0, a floor tile. In English -

the player finds a sack of gold and picks it up - the sack disappears revealing '5000 points' which stays for a given time then vanishes to leave a bare piece of floor. All this can be done with the minimum of coding. This is one example of the command Tile Val.



TOME also gives you the ability to change palettes which with a bit of forward planning could let you move from day to night time, using the same graphics.

Once your world has been created, it is saved to disk and is then Bloaded into memory. The map is Bloaded into Bank 6, and the tile val files are saved out at the same time as the map and are Bloaded into Bank 8. Bload is AMOS's binary loading method.

When you have created your map, you can scroll the world by using one of the routines supplied by Aaron Fothergill. It would be quite a task controlling a large world if you had to do it all yourself, but you have the choice of horizontal, vertical or 8-way scrolling given to you. Marvin uses 8-way scrolling so that it is easier to move around the maze.

There is also an AUTOMAP feature which creates maze type maps, or at least goes as far as doing 75% of the work for you. After choosing the size of the map etc from the options given, TOME will draw a map for you. All that's left for you to do is tidy up the result to suit your needs. Be warned though that a large map will take a long time to create, so go make a coffee or catch up with the latest video while it's working! Just think how much longer

it would take to create it manually! In the Automap feature you follow the instructions for placing the tiles in their correct places so that your left turn matches TOME's left turn etc just follow the prompts for choosing the other options, then sit back and wait!

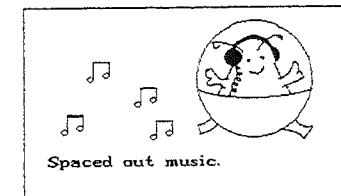
At the time of writing, this extension is the closest thing to a construction kit that AMOS has, it leaves your time free to check on tile values, and put in the options and conditions necessary to complete the game. There is help available on screen if needed, and the full manual is on disk.

There are so many different types of games that can be written using TOME that anyone who wants an easier way of game writing in AMOS should seriously think of adding TOME to their set up. One of the biggest advantages of using TOME is the fact that it saves so much memory when you're writing a program. If you had to write routines that did all that TOME does for you, your listing would be a lot longer!

CHAPTER FIVE

MUSIC AND SOUND EFFECTS

No game is complete without music and sound effects. Much of the atmosphere of a game is conveyed by sound and taking the time to find the right noises is time well spent.



A collection of Public Domain Disks containing sound samples can be invaluable if you do not have a sound sampler of your own. Several good examples can be found in the Deja Vu PD Library.

The opening music in Marvin the Martian is a piece of classical guitar music which was put into SoundTracker and converted to Amos format with the SoundTracker convertor.

We used to wonder why sound samples interrupted the playing of some music but not other pieces. The ones we'd put in ourselves were almost always O.K. while the 'heavier' styles of music stopped and started for the sound effects.

After looking at both styles in Sound Tracker, we saw the answer. Our music rarely uses all 4 channels, therefore by using the spare channel for samplaying, the music carried on. So if you can keep a free channel, do so.

To make the sam play on channel 4, use the command:- `Samplay8,sam number,frequency`

The default is channel 1.

Here's a list for all 4 channels, using sam number 1 and frequency of 10,000

as examples

To Play sam on channel 1 - Samplay1,1,10000

“ “ channel 2 - Samplay2,1,10000

“ “ channel 3 - Samplay4,1,10000

“ “ channel 4 - Samplay8,1,10000

Using this you can have music and sfx at the same time.

If you are thinking of putting in your own samples then this can easily be done by purchasing a quality sampler. Beware of buying the cheap makes as these are usually not very good and this can be detected in the sound quality.

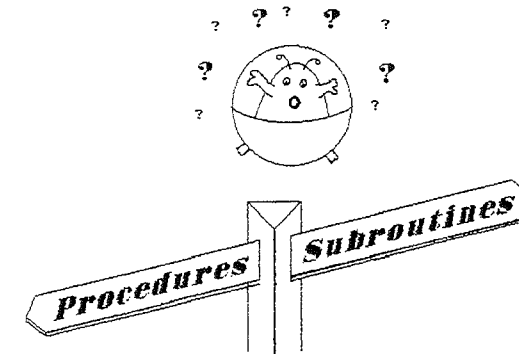
Reading Amiga Format is the best place to go as they have regular features on items such as sound samplers, expect to pay anything from £30-ove £1,000.

The TFMX soundtool from the software business is the most professional sound product on the market based on using samples. It costs around £45, but using a PD program such as SoundTracker is good enough for most purposes

CHAPTER SIX

STRUCTURING YOUR PLANS

Something you should keep in mind as you write a program is that you will, more than likely, need to be able to follow the listing at a later date to correct errors, this is known as debugging. No program can be written without some sort of debugging being necessary. There will always be something that needs changing whether it is the speed something moves or a major fault that appears when the program is being run.



When it comes to deciding which way to write your program, choose the method you like best.

One thing to decide is how you are going to write your program. There seem to be two main methods, one using mainly procedures, the other subroutines. It does not mean that the two cannot be mixed to a certain extent, it's for you to decide which method suits you best, it helps to be comfortable with the way you write. We use procedures as this method suits us.

To help you decide, here are a few points for and against each method.

When you use Procedures, you can put routines into self contained sections inside your program, eg a print routine and give these sections names which you can call from another part of your program to perform their function. If thoughtfully written, these procedures can be used in other programs just by cutting them out as a block and merging them into the new program. Your listing will be shorter to search through as you can 'fold' the procedures

which means that you can hide the listing of the procedure so that only the procedure name is visible.

Giving the procedures descriptive names will help you to identify what each one does. It is surprising how much time this facility can save when searching through listings. Try this with the uncompiled listing of Marvin. Load it into Amos and scroll through the listing taking account of the time it takes to reach the end. Now unfold the procedures and repeat the scroll through the listing. Do you see the difference?

One other thing, if you are not compiling the program, procedures can be locked to 'protect' your coding from prying eyes. but be warned, you cannot unlock a locked procedure! You must always keep a source copy of your work in case of alterations. The same goes when compiling a program, keep an uncompiled copy!!!!

If you use procedures, you will have to have global or shared variables when you initialise your program. This is so that the procedure has access to the values, but this is not necessary for variables which are exclusive to a specific procedure. Global is used at the initialisation of a program, and Shared is used inside the relevant procedure. Also there is a very slight slowing down in the running speed of the program, but this is nothing to worry about.

If you choose to write using subroutines, each part of the program is put into a subroutine and you access them by using Goto or Gosub and return. Debugging can be more difficult as the listings are so much longer. The listings are lengthy because you cannot hide any lines as when using procedures, there is however a very slight increase in running speed and there is no need to use global or shared when initialising.

Marvin, as you will see, is written with procedures, but when we were having trouble compiling the program, all the procedures were taken out and subroutines used, it did not make any difference to the look or play of the program, no one using the program would have seen any difference.

The only thing to remember is that you should be able to follow the program and understand what is going on days, weeks or even months after you wrote it!

CHAPTER SEVEN

BRIEF TOUR OF MARVIN

This is just a brief description of how Marvin flows. It will probably help if you are looking at the listing as you read this.



We start off by initialising the program, this means that the variables are listed and given the values for the start of the game, these include things such as the energy level and the number of levels etc. There is also a list of global variables so that the procedures can access the values these variables hold.

The program first goes to a procedure called FRST which loads in the panels and displays the credits. Underneath this is a label NW: which is where the program comes when the player is GAME OVER.

Next comes a list of initialised variables, these are set to default on GAME OVER. They include the level number, the number of coins, keys, lives etc

The procedure FRONTBIT follows, this loads in all the information needed for the options screen, this includes the help screen and other options. These are dumped from memory when you go into the game to save space. A secondary loop allows you to put in a password.

Procedure INIT loads in all the initial data and sets up the screen displays etc. If you lose a life, but it's not GAME OVER, you come back to the label STRT:

The next part of the listing initialises the second set of variables. These keep count of what state the game was in when you lost a life.

BNUMBERS procedure contains a part of the beast data, also the bonus data and password data. INIT2 loads in the relevant information after loss of life or for going onto next level. If you have earned enough coins to gain an extra life, this is checked on here, if a new life has been earned, it is placed somewhere on the next level.

Now we start the main game loop. To save memory, another tip is to give variable the value of Free - this gets rid of unwanted bits and pieces and reduces memory leakage, which is a fault in the design of the Amiga, not Amos.

First of all, the program checks to see if you are dead, if you are, there's no point going any further! If you are still alive, it checks to see if a level has been completed. The variables are set ready for the TOME map routines and the map coordinates are checked to see if the map has moved since the last pass of the loop, then the old variables are set.

Now there is a check on the objects on the map that use a counter, for example, things that are placed on the map to be picked up. In this program Marvin picks up a harp, the harp disappears and a number is displayed for a short while before it is replaced by a floor tile making it look as if the harp has been picked up.

If you kill an enemy, a random routine comes in which decides whether or not you get a coin as a reward, if the answer is 'yes' then a coin is printed on the map.

The following line copies the altered work screen to the physical (ie the display) screen. Then the program forces the maze to be displayed at the start of the game and puts Marvin in the right place.

As we have an energy level this is checked on and the relevant action is taken. For Example, if the energy bars have run out, then a life must be deducted.

The loop then checks to see if there has been a collision between Marvin and any of the enemies, if there has been, we go to the BSTDEAD procedure.

The physical or display screen must be synchronised with the logical screen

this is done next. Also the joystick must be made to stop when you stop, this is done by resetting the joystick variables.

To kill an enemy, Marvin drops an oil slick, this is controlled by pressing the fire button. If you are alive, ie you have some energy, you are allowed to move the joystick and use the fire button. If the fire button is pressed, then a slick is planted. The slick also works on a timer, when a certain length of time has passed, the slick is replaced by a floor tile.



The procedure CHKPARAM checks for the parameters on the display screen and the map coordinates to make sure that Marvin stays on the viewing area of the map and that the map does not move beyond the limits previously set. It would not look right if Marvin was allowed to wander over the control panel! The procedure called CHKWALLBUMP detects a collision between Marvin and a wall. If any information has to be updated, then the Procedure PRNTIT prints it!

DNR: is a label, the program comes straight here if the player dies so that there are no ghost movements caused by going through the rest of the loop.

WEND ENDS THE GAME LOOP

This main loop checks for any information that needs to be updated. As the loop runs continuously, the information is updated as soon as is necessary.

The next routine comes in if all lives have been lost, it clears the bobs screen and slides on the GAME OVER bob. If the level that has just been completed is less than level 15, the screen slides off ready for the next level. If the level just completed is number 5 or 10, a new set of graphics is loaded, the size of the maps increase after these levels.

If the player has earned enough coins to get an extra life, it is dealt with here.

If the player has more than 19 coins, a message is displayed to inform them that they must now claim their extra life by finding it on the next level.

When a level has been completed, the player is told the password which will allow him/her to restart at the new level without having to complete the preceding levels again.

If the level has been completed and the player is still alive, the end sequence is displayed before the game is taken back to the start point.

The procedures are stored at the end of the program's main listing, this is where we prefer to keep them, all together so that they can be folded and easily found when needed. They could just as easily be put at the beginning of a program if you prefer. You can put them into the listing as they occur if you wish, but this will make them harder to find in a long listing.

This is a very brief description of the procedures in the order they occur at the end of the main listing.

FRST This loads in the panels and the display for the credits, it also grabs blocks to use as masks.

SVE This procedure saves the high score variables to disk after asking the player if this is required.

EBIT is the end bit, that is the reward screen with animation shown if the game is completed.

DEAD is called if the player has died.

FRONTBIT This loads in all the initial information including the help screen and high score tables which are dumped as soon as the game is entered to save memory. A secondary loop allows you to enter a password to start at a higher level.

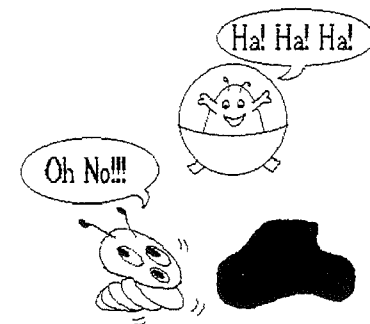
PSWRD The password input routine has its own loop in this procedure.

SDOWN[SPEED,SOURCE,DEST] This is how the intro screen is brought into view, it can be cut out to be used in other programs. This is on

advantage of using procedures.

WIPE This routine can also be cut out for use in your own programs, it is the screen wipe that takes you into the game from the options screen, it looks as if the screen is being wiped by the hands of a clock.

BSTDEAD This is called if an enemy dies by bumping into Marvin.



You do not have to use mega-blasting guns to get rid of your enemies.

KLLSLIK[BNM] If an enemy dies through slipping on an oil slick, this procedure is called.

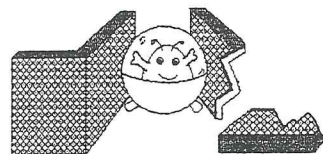
DMPSLIK This puts a slick on the visible area of the map and starts the timer which determines the length of time it is displayed.

PRNTIT This procedure is called when information needs to be updated on the screen. This covers such things as the displays of the score, the energy, items collected etc.

CHKTILE This works out which tile Marvin is standing on and what should be done as a result of the information.

PTLIFE This puts an extra life somewhere on the map of the following level if enough coins have been collected.

CHKWALLBUMP This tells the program how to react if Marvin bumps into a wall.



The procedure **CHKWALLEBUMP** checks if Marvin has bumped into a wall.

CHKPARAM This checks for the parameters on the display screen and the map coordinates to make sure that Marvin stays on the map and that the map cannot be moved farther than intended.

SETUP sets up the map variables and the speed at which it moves.

NICEMAP This routine is supplied by Aaron Fothergill on the TOME disc; it moves the map around.

INIT2 This loads in the relevant information needed after a life has been lost or when going onto a new level.

INIT The initial data for the program is loaded in by this procedure, this includes the screen displays.

STRTLE[P,PX,PY] This gives a value to the dimmed array which covers the occurrence of more than one bonus item being put down at any one time.

PTTILE This routine is needed if a tile showing bonus points has been placed on the map. It is displayed for a set time, then it is replaced by a floor tile.

OTHERBLOKE This is the bit of the program that controls all the activities of Marvin's enemies wherever it occurs in the program.

RANDOM[ANMT] As the name suggests, this is the random generator which is used throughout the program.

NWBST[LK] There is a set number of enemies which can be generated

any one level. This procedure regenerates an enemy if this level has not been reached. This means that a new enemy will replace one that has been killed, but as the game progresses, less and less enemies are produced so if you keep on killing them, there will be less to worry about as you get to the end of a level.

BSTDTA This routine decides where to place an enemy at the start of a game, and also which direction he will be facing.

BNUMBERS This contains part of the data needed for the enemies, the bonus data and the password data.

SCNON[SCR] This slides a screen back into view after one has been taken off.

SCNOFF[SCN] Takes a screen away.

RAIN This creates the rainbow effects seen in the high score section. It reads in the data created by **RAINBOW WARRIOR**. (A PD program by Martyn Brown, as described earlier.)

SRT The high scores are sorted into order by this procedure.

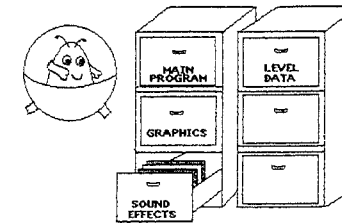
WARN[M\$,B\$] Here is the procedure that creates the alert boxes seen in the game, that is the requester screens that pop up if it is necessary to check on something that the user has to do. e.g. Are You Sure? YES or NO. These are affectionately known as idiot detectors as they can be used to stop the user doing something he/she'd regret later on.

That's your brief guide to Marvin the Martian! It should help you to follow how the program has been laid out, the following sections will deal with how the different effects and routines are done.

CHAPTER EIGHT

MARVIN'S PROCEDURES

It would be very difficult to cut up the listing of Marvin the Martian into tidy little sections and explain how it was written a line at a time. So, we'll take you through the program by explaining each procedure or loop as it occurs.



Organise the files on your disk so that you will be able to find them easily.

It makes sense to organise the files that you will need so that you will know where to find them when they are needed. If you store every file in the root directory of your disk, this takes a long time to look through and there is more chance of a mistake being made when deleting files. Create separate directories for your graphics etc and keep the root directory for things that you need to access quickly. We stored all the data for the game levels in a directory named 'lvls' so that there was no doubt as to what the files contained. The directory 'gr' holds the graphics, ie the bobs, screens, tiles and panels. It would be a good idea to store all your sounds in their own directory as well. If you do, for example store all your graphics in a directory called 'gr', don't forget to put 'gr/' in front of the filename when loading it into your program, or you will be greeted by a 'file not found' error message!

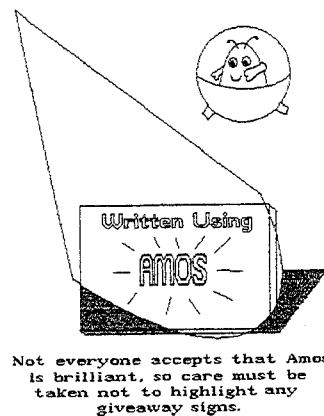
The first line of your program should always set the size of the buffer. Amos reserves a buffer size of 8k for the variables used and the work that they have to do. If this buffer space is too small, you will be given an error message 'out of string space'. The manual suggests increasing the buffer by 5k at a time, but if you are writing a big program, 10k may be more practical. Do this until the error message disappears. If the buffer size gets too large for the available memory, you will have to try and save memory elsewhere in your

program. You have to take into account the amount of memory you have the developer of the program, and also the amount of memory owned by the intended users of your game. It may be that you have 2 megabytes to play with, but if all this is used, your potential market will be greatly reduced. Everyone has 1/2 meg, many people will have 1 meg by now, but there are very few with more than this.

The next line is always at the start of our programs.

```
Screen Open 0,320,200,16,Lowres: Curs Off: Flash Off: Cls 0
```

Even though Amos defaults to screen 0, it is always good practice to open a screen here and control the display for yourself. This is because there could be problems when you come to RAMOS your disk or compile the program. You don't want to see that horrible orange screen at the start of your game, so open a screen. The next three commands are always the same, Curs Off: Flash Off: Cls 0. If these are done in this order you won't have a ghost cursor in the top left of your screen, as seen in so many Amos programs. It is said to be a bug in Amos itself, but it is so easily cured that it cannot really be counted as such. Just make sure that the screen is cleared to colour 0 last line (Cls 0).



If the last three commands were not used, you would get the orange Amos default screen with the cursor flashing on and off in the corner of the screen.

This is a giveaway signal that Amos was used to create the program.

Whereas many software houses accept Amos as programming language, others unfortunately see it as 'only Basic' and will not look much further. One of the purposes of this book is to help programmers make their programs look more polished, and so more acceptable as a commercial release.

You will notice that we open a screen to only 200 pixels deep, not the full depth of 256 pixels of a PAL screen. This is because American and Japanese screens are 200 pixels deep, and if you are serious about getting into the games market, it would be wise to get used to using this screen depth. This size makes the program suitable for use on NTSC machines.

Another tip is to use 16 colours wherever possible, this sounds rather limiting, but with a bit of planning lots can be done. You can give the illusion of more shades by filling areas with pixels of two alternate colours, like a chess board, grab a block of 4 pixels as a brush and use this to fill the area. Gold and white gives a flesh coloured effect.

Having said that, Marvin does have 32 colours, it was written well before we realised the benefits of fewer colours and converting down to 16 colours was not a realistic move. It not only frees more memory, it will make any game like Marvin move faster, even when uncompiled.

The screen opened in the first line is just as a default setting for the program, all other screens are loaded with 32 colours.

MEMORY SPACE BOOKING OFFICE					
Zone	Zone	Variable	Variable	Bank	Bank
SFX				Amos	Graphics
				Amos	
	Text			Amos	Text

There are times when memory space has to be booked in advance.

The Dim line forces Amos to allocate memory to hold important arrays used inside Marvin. These arrays include the High Score names, there are 14

allowable entries, the High score numbers, again there are 14, the position of the enemies, their animation frames and the alive/dead status of the baddies.

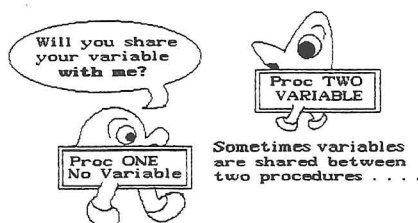
For Example:- Dim MORT(40) MORT is the variable used to keep track of all the enemies. Initially all these are set to 0. If an enemy dies, the MORT number corresponding to the (dead) enemy number is set to TRUE, so that there is no need to check on him again when doing enemy movements.

The excessive use of GLOBAL is often frowned upon by programmers who say that Amos doesn't like it being used. They may be right, but we've never had any problems ourselves, and as you can see, there are loads of Global variables in this listing! If this program had been written now, there probably would have been less use of Global, but as the saying goes, if it works - use it!

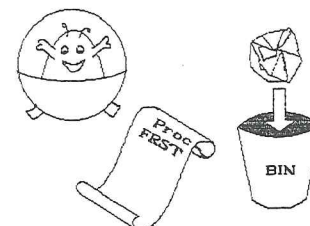


Most variables have to be Global.

Using Global does make life a little easier in that you do not have to remember to use SHARED inside a procedure. A good rule to follow is, If a variable needs to be accessed by many procedures, use Global. If a variable is needed in just one or two procedures, use Shared. If you do use Shared variables, remember which variables have been used.



Procedure FRST. The first procedure we come to is aptly named FRST. It is only called once, and then promptly forgotten.



Procedure FRST is used just once, then put in the bin.

It loads in the Credits screen which it stores in Bank 12. There is no specific reason to use this bank number, the only ones you cannot use are those which have been reserved by the Amos System, so when needed for data, take your pick!

Next it loads in the panels screen, this contains the graphics for gadgets used by the player to enter a password and their name for the high score table, and a large graphic that says 'GAME OVER'. It is unpacked from bank 14 to Screen 1. Screen 1 is then hidden so that no one can see the scenery being shifted.

THEN WE HAD A CHOICE TO MAKE

1. The images could be left on the screen and the program could have worked on them from there.

or the option chosen to save memory

2. The images could be grabbed as blocks and pulled back as if out of fresh air when needed.

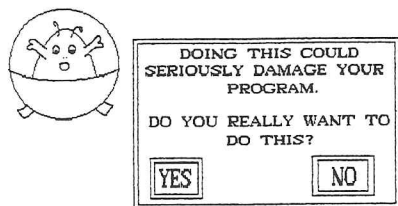
Bank 14 was then erased, thus freeing memory for use elsewhere in the game and screen 1 closed for the same reason. Remember, the more screens that you open, and the more unnecessary banks that are left in memory, the more memory is gobbled up. This will lead to the dreaded 'out of memory' error

message.

Next you are sent to the procedures HI and SRT which will be described later on.

Procedure SVE: This procedure is activated under two conditions - one is the options screen. If 'S' is pressed, then the program is sent here, or if the High score chart alters, then it also comes here.

The first thing that happens is that the program goes to the procedure WARN. This is an alert box routine that asks if you want to save to disk. It checks the parameter variable 'q', if q=1 then you want to save, if it equals anything else, then nothing is to be saved. This is an example of an If....End statement. The save will only take place IF the condition is true i.e. if q=1. q does equal 1, then we go to WARN again to make sure that the disk is write enabled.



An alert box can warn the user that he or she might be about to make a silly mistake

Another tip - Don't allow system boxes to appear, eg 'disk is write protected' as it is not 100% guaranteed to let you go back to your program on pressing the 'retry'. Make sure that you are in control of the Amiga, don't let it control you!

You've done all you can to warn the player that the disk must be ready to save, so if he goes ahead with a write protected disk, then he is to blame for the consequences!

To save, you need to open Channel 1 out to the disk. This opens the file called high scores.s. This file already exists as there is a dummy file saved to fill up the score table when the game is first loaded. The data is stored on

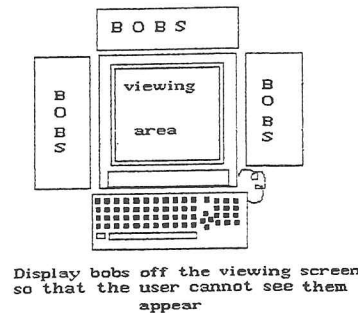
the disk using a For... Next Loop. The information is then written to the disk, the name and the relevant score. The channel is then closed and the program continues on its way.

EBIT is the procedure that controls the reward sequence if all 15 levels have been completed, hence E(nd)BIT. The old display is cleared away and the player is allowed to enter his/her name, if appropriate.

The game screen is then brought back and banks 1,2,5,6,8,12 and 13 are erased. A new screen, number 1, is opened, don't forget why Curs Off : Flash Off : Cls 0 come next in that order. Then the program gets the palette from screen 0. The reason for opening the screen is that Screen 0 is Double Buffered and the only way to switch Double Buffer off is to shut down the screen. This will free the memory it was using. Now we load in the end sequence itself, the bobs and the sound effects. The picture for the end screen is unpacked from bank 15 to Screen 0 and Bank 15 is erased. Now Screen one is flicked to the front as we are not ready to display the end screen yet. This has to be done as unpacking a screen forces it to the front so Screen 0 is 'appeared' to Screen 1 here, then Screen 0 is shut down.

Double Buffer is needed again as there is an animation of Marvin and Meryl flying away. Bob update is turned on as we don't need any fine control for this part of the program as we did in the main game - it's more like a demo where things happen automatically. We use AMAL to do the animation as the bobs follow a set path across the screen. Four AMAL Channels are set up and the four bobs are assigned to them in a For Next loop to save time.

The Bobs are positioned slightly off the screen to the left of the viewing area so that they are out of sight. The first part, Anim 0, continually loops. The numbers in brackets are the image number you want to animate and the length of time it is on the screen. The Move 500,-200,300 line is explained as follows. 500 moves 500 pixels to the right, -200 moves 200 pixels up at the same time at a speed of 300. The same motion is used on all four bobs. Amal is turned on as Amal On 1 : Amal On 2 : Amal on 3 : Amal On 4 : This could have been done with just Amal On instead of turning on each Channel individually.



Sam Loop On continually loops the sound sample which is played using Sready to print the user input. Play 1,1,8000. This plays on Channel 1, Sample number 1, at a frequency 8000.

We next have a delaying loop - While Chanmv(1) : Wend This means while Channel 1 is doing its move, loop, - ie keep the sound going until Bob has gone.

Amal is now turned off, as is the sound, Bank 5 is erased and the screen shut down. This completes the sequence.

The next procedure is called DEAD. This controls what happens when you die because you've run out of lives.

It first checks to see if the score gained is higher than the lowest one currently displayed on the high score table. If it is, then a new screen is opened, a smaller screen this time to save on your precious memory, and a block containing the graphics for the input device is displayed on it. The lowest scoring name is removed from the high scores list and the list is re-sorted to the new high score. The information is then turned into a string. There is a reason for using a string rather than a numeric variable, this is that all scores are stored as six digit numbers and in a score of less than 100,000 there will be zero's preceding the score. eg if you get 1000, it will be stored and displayed as 001000.

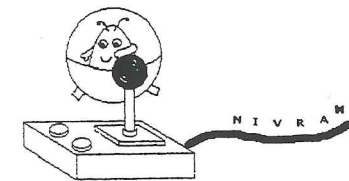
The display used to input the player's name works as follows. There is a picture of a device that resembles a remote control gadget with a display panel for the information entered. The idea is that the name is input in one of two ways:-

1. By mouse clicking on the arrows to move up/down the alphabet until the correct one is reached when the 'enter' button is clicked on. The name is entered either when the display is full, or the 'end' character is selected.

2. The joystick can be moved left or right to move through the alphabet. The fire button selects the letter.

Now we reserve zones for the buttons on the input panel and change the ink ready to print the user input.

Yet another loop checks on the mouse zones. Here we have Mouse Click, not Mouse Key. The reason for this choice is that Mouse Key repeatedly checks for the Mouse button is being pressed, Mouse Click just checks for the one key depression. This is useful for checking on once off detection.



Is this Joystick being used to write a name for the High Score Table?

There also has to be a check to see if the joystick is being used to input the name. The routine looks to see if the joystick is being moved left or right. There is a delay put in here so that things are done at a realistic rate, the letters must not move so fast that you cannot read what they are!

This part of the program loops around printing the letters that are selected and adding them to a string until the 'end' character is selected.

The scores are then sorted into numeric order and the player is asked if he wishes to save to disk. If he does, then it is saved, if not, the program goes on. Note that you can save by pressing 'S' on the options screen as well.

Procedure FRONTBIT is the next on the list. This controls the options screen. A screen is opened in the now familiar way. You should know this by heart now!

The procedure checks to see if there is any music in memory - if there is, it plays it. If none is in memory then it loads it from disk, then plays it. It does this by checking on the length of bank 3 which is where the music is stored. If the length of the bank is greater than 0, then there must be some music there, if the bank length is less than 0, then it is empty and so music must be loaded.

The packed help screen is loaded from bank 13, and the packed title screen is loaded to bank 15. The title screen has been 'protected' so that it cannot be unpacked in the normal way. There is no reason for this screen having protection, other than that this was the program being worked on when I learned how to do this, it was tried as an example, but not taken out again. you get an extra tip for nothing!

This is how it's done. Title.pac is loaded into bank 15, but if you try to unpack it you will get 'not a packed pic' error message.

Try this for yourself to prove it. assuming that the disk with Marvin is in Drive 0 go into direct mode and key in Load "Df0:gr/title.pac",15 and press Return. You should now have a packed pic in Bank 15. Press Left Amiga + F1 to show the bank list on the screen and you will see the bank contains the title.pac file. O.K. Now type in Unpack 15 to 0 and press Return. Now you will get an error message.

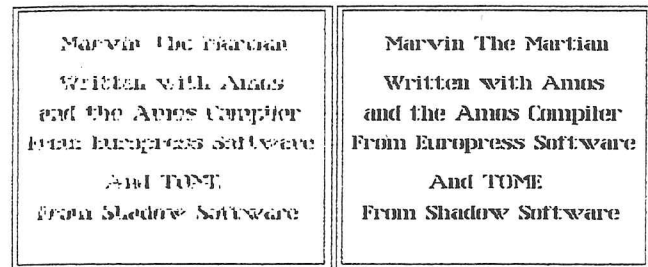
So how do you 'unprotect' the file? Before you try unpacking the picture, in this simple command Poke Start (15), \$12 and press Return. Now you can unpack as before and it will be unpacked.

This little bit of magic happens because when the graphics were being prepared, after packing the screen to 15, this command was put in, Poke Start (12),0 This is a very simple form of protection for your graphics, although not quite as safe now that we've told you how it can be done!

The title screen is unpacked and displayed off the visible screen (hovering just above the monitor to be precise!) As in the loading of the music, the Length command is used to check if something is in Bank 12. If the Bank length is greater than 0, then its contents are unpacked to Screen 5. These are the credits which only appear on the first pass of the game. It can be a source of annoyance to be forced to go through unnecessary screen

before another try at the game.

The whole screen is grabbed as a bob and Bank 15 is unpacked to Screen 5. Now we display screen 5 off the viewing screen and the bob with the credits is pasted onto it. This bob is deleted immediately so that the memory it is taking up is released again, a bob of this size takes up quite a bit of memory. This sets up the screen for the appear effect when the credits are being



By Creating a second, identical screen with the credits added, you can use the Appear command to give the illusion that just the words are pixel appearing on the screen.

Displayed on the screen.

This effect is very heavy on graphic memory, so it's best not to repeat it every time the game is replayed. Bank 15 is erased, screen 0 is selected and the program gets the palette from screen 6. The routine SDOWN is called, this is a parametered procedure which has three parameters, SDOWN[A,B,C] where A=speed of effect, B=source screen and C=destination screen. The screen effect is described later.

Now we come to another condition which finishes off the effect. If the length of bank 12 is greater than 0, then the program is instructed to use the appear command to make screen 5 appear through screen 0. As the only difference between the two screens is the credits display, this gives the effect of just the credits appearing on the screen. Screen 5 is now erased. Remember - if you don't need it, shut it down!

Next a delay loop makes Amos sit and wait until a certain condition is met. While Mouse Key=0 : Wend ie do nothing until the mouse button is pressed.

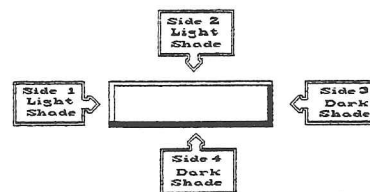


Sometimes you have to make AMOS sit and wait for the Amiga to catch up!

To make sure that the appear effect only happens the once, Bank 12 is erased therefore the next time the program comes here, the length of Bank 12 will be less than 0, ie empty, so the effect will not be used. The credits are removed from the screen by copying screen 6 to screen 0. Screen 6 is similar in appearance, but it has the buttons for selecting the game options on it.

The buttons now have to be created so that the player can choose what he/she wants to do. This is done using the Cls command followed by the Draw command. To make these buttons a bit more interesting than just boxes on screen, they are drawn so that the illusion of 3-D indenting buttons can be created.

The lines have to be drawn in a certain order and in certain shades so that the effect can be achieved, this is shown in the following diagrams.



The borders for the buttons must be drawn in the above order and colour to give an indenting effect.

To make these buttons active, zones are used so that the program can detect which option the player has chosen.

The zones are reserved, and then set to the relevant co-ordinates. Now

have a loop, this time we are checking for three things inside the loop. These are, a mouse press, a mouse zone and a keyboard press.

For the keyboard input, I\$=Inkey\$. The first input checked on is for the letter 'M'. This is not a straightforward check as 'M' is used to switch the music on and off, not just to make a single choice. This toggle facility is created with the variable MO.

'S' is the next keyboard input checked on. This is the save facility for the high scores which can be saved here by pressing 'S' as well as after the game has ended.

All the buttons that were created with the Cls and Draw commands are scanned to see which one is being used. As we mentioned before, we wanted these buttons to be indenting, it's an effect that looks good and is easy to create. This is done in the five subroutines that are at the end of this procedure called BUT1 to BUT5. Rather than using a routine to call these five subroutines, the mouse zones are put inside a variable, (this saves a lot of keying in!) this is very easy to do and once done it is easy to create a routine that will check on all the buttons at the same time.

With the first method, it would take five routines to call the five subroutines, with the one we used, it takes just one - MZ=Mouse Zone If Mouse Key=1 and MZ>0 and MZ<6 On MZ Gosub BUT1, BUT2, BUT3, BUT4, BUT5 End If This works well with Procedures as well as Subroutines - instead of On MZ Gosub, put On MZ 'Proc'. Remember though that the procedures or subroutines should be in ascending numeric order.

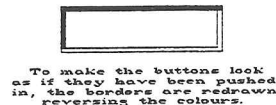
This controls the whole front end, the rest is a subroutine. Now that we are out of the loop, we must clean things up and get out of the procedure. Erase banks 13 and 15 and make the screen disappear with Proc WIPE. Here the command 'Pop Proc' is used. This now seems to be an unpopular command to use as it can give errors, but it seems to work here. If this program was being written now, we'd put Goto OUT here, and on the line immediately before End Proc put a label OUT:. This will do exactly the same as Pop Proc. The program jumps to the Label, then carries on to the End Proc and you're out of the Procedure.

Even though you are out of the Procedure, it's work is not finished. Now

come the routines for the five buttons, the password, help and the highscore

The subroutines for the five buttons do the same thing, but with different ordinates.

The buttons have to be re-drawn so that they look as if they have been pushed in when the Mouse button has been pressed over the zone. This is achieved by switching the colours on the button borders, the two sides that had light coloured sides are now dark, and vice versa.



We have used a While Wend loop here so that the user can see the effect of the indented buttons. So, while the mouse button is pressed, the buttons remain in the 'pushed in' state, when it is released, the borders return to their original colours giving the effect of the button being released.

Button 1 - The password subroutine. This is in the routine called BUT1. When this button is pressed, the program goes to the Procedure PSWD, which will be explained later.

Button 2 - The Highscore subroutine. This is contained in BUT2. When the mouse button is released, the procedure SCNOFF[A] is called. This is a parametered procedure in which A = the screen number, which is 0 here. The screen slides off and will be explained later.

Now that the options screen has gone away, screen 1 is opened, the palette is grabbed from screen 0 and it is displayed off the viewing area, you don't want the user to see what goes into setting up your display. Now the palette and pen colours are set up and the list of high scores is printed in order, when screen 1 was opened, note that the line did NOT contain the usual 'Flash' command, this allows us to make the top score flash to highlight achievement.

Now we go to the procedure RA1N. This sets up the pretty tubes on the screen behind the names on the high score table. This was created using Rainbow Warrior (see the information at the beginning of the book). Once activated, the screen is brought on using the procedure SCNON[A] where this time the A=1 for screen 1. A While Wend loop keeps this screen in view until the mouse button is pressed.

To save memory, screen 1 is shut down, remember to do this to screens that are no longer needed unless they have to be loaded from disk.

Rainbow Del 0 is a command that you will not find in the manual, it turns off the rainbow effect on screen 0. Now we call SCNON[0] to make screen 0 the current screen.

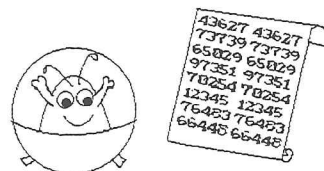
The next button is used just to display the current game level, therefore the subroutine BUT3 does nothing, it was left in here not to upset the flow of the gosub routine.

BUT4 controls the forth button which selects the help screen. The very handy SCNOFF[] procedure takes the options screen off, bank 13 is unpacked to screen 1 and it is displayed off the viewing screen. Then SCNON[1] is used to display the help screen which is told to stay displayed until the mouse button is pressed again by a While Wend loop.

The final button is controlled by BUT5. This is the START button which lets you into the game. All this subroutine does is set the condition STRT to true, this is the only condition needing to be fulfilled to exit the options screen.

Procedure PSWRD. This looks after the password facility. You cannot save your game position in Marvin, but there is a password to be earned at the end of every level, this will let the player enter a higher level without playing all the previous levels.

Here we find an example of a Shared variable. Shared NBR tells the Amiga to share this variable with the previous procedure, ie FRONTBIT, it will not share it with any other procedure. SCNOFF[0] gets rid of screen 0. Screen 1 is opened, and block 9 is put in its place on the screen. This block contains the graphic used as a device for entering the password number.



The password is checked to see if it will get the player onto another level.

As this panel has clickable buttons, zones now have to be set so that the program knows where the mouse button has been pressed. Rather than use individual Set Zone commands, we used data statements to store the coordinates - there are four sets of coordinates per Set Zone command. The a For Next loop is used to read in the data. Another loop is used to create the indenting button effect and to check on the mouse zone and the variable in the zone.

INDY1 - INDY10 are the number buttons and ENTR is the enter button as on a calculator. The only things stored in subroutines are the coordinates to draw the buttons to.

Once the number is entered, the program checks in Proc BNUMBERS to see if it is a legal entry, if it is wrong, a message tells the player that he/she is wrong and the routine is exited. Our handy procedure SCNOFF[1] is used again to get rid of screen 1, SCNON[] brings on another screen and the loop is exited.

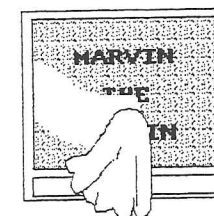
Once out of the loop it checks to see if the correct password has been entered. If it has, the display is changed on screen 0 to the next level. The variable used is LVL. Pop Proc takes you directly out of the procedure, but as we saw before, this is not a command that we'd use now.

Indy1-10 and ENTR hold the drawing coordinates for the buttons.

Procedure SDOWN[A,B,C] This is the parametered procedure that was mentioned earlier. A = speed. B = the source screen and C = the destination screen. This is a procedure that can easily be taken out of this listing and used in your own work. This procedure cuts the source screen into horizontal slices, each slice being the height of the speed variable. It then slides the

slices from the top to the bottom of the destination screen, which gives a melt effect. The only difference between this routine and a standard melt effect is the line which starts If Source = 6. This checks to see if the user presses the mouse button while the credits are being displayed, if so, the credits are not displayed. This will save the player time if it's not the first time that the game has been played and he/she does not want to read the credits again. This feature is often seen in commercial games, where by pressing Fire or clicking the mouse button, you can skip over the title screen etc to get into the main game. If you wish to use this procedure in another game, then make sure that this line is deleted.

PROCEDURE WIPE



Using different screen wipes to change the screen display will add interest to your game

This is another effect used to change the screen display. It makes the program flow better if you use a wipe or melt effect to switch between screens rather than just swapping screens. This effect is seen when you press the 'Start' button on the options panel. The effect clears the screen to black and it looks as if the picture is being wiped off the screen with the sweeping hand of a clock. This is done by drawing black lines with the Polyline command.

Again this routine can be easily cut out and used in your own programs.

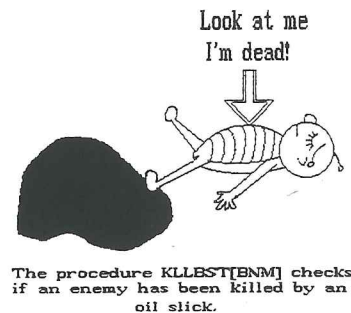
PROCEDURE BSTDEAD.

This procedure checks to see if a beast (enemy) has died during the game. The program follows a few steps to do this.

The majority of the procedure is inside a For Next Loop - For Z=1 to

BSTINLVL. This checks for all the enemies in the current level. The number of enemies increases as the game progresses through the levels, we decide to do this for two reasons, one, a game should get harder the higher you go and two, because the map size increases. If the number of enemies stays the same, it would in effect look as if there were less enemies to get rid of in higher levels.

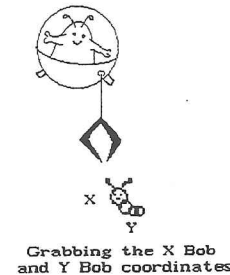
The whole procedure is called from the main program if an enemy collides with Marvin.



The routine finds out which beast Marvin has bumped into. On initialisation each beast was given a number, and to find out which beast has been hit by Marvin, we use the above For Next loop in conjunction with the CENEMIES command. Col(2) will hold a value of either true or false, it will be true if there has been a collision between this Bob and Bob1, which is Marvin. The same routine cannot cover a beast's death if it collides with a slick as oil slicks are tiles, (which are icons) not Bobs.

Inside the For Next loop is the If End If statement If Col(Z) This is short for If Col(Z)=True, True is automatically assumed unless the program is otherwise. If collision is true, then a flag variable is set saying that a beast died, this lets the program know that there has been a valid collision.

Next, the coordinates of the dead beast are grabbed using the Xbob and Ybob commands. Bob Off(Z) now turns off the beast's bob. Now again there is an If End If inside the last one.



In order to describe the listing further, we will have to explain the way in which the enemies are organised. To create a playable game we worked out that we would need to have more enemy bobs on the screen than Amos could comfortably cope with. To get the required number of enemies, we control each bob with two variables.

This allows each bob to be re-used to give the illusion that there are more enemies lying in wait for you than there are bobs available. To put this another way, it's as if each bob has X amount of lives, if the bob loses all these 'lives' then the bob is switched off. Therefore as you play the game, you will have plenty of enemies to kill, but as you get near the end of a level, you will notice that the enemies get fewer and fewer, this is because more and more bobs have lost all their lives and so do not produce more enemies.

The program checks to see if the beast limit is still valid, if it is greater than 0, then it goes to the routine that creates new beasts this is NEWBST. MORT(Z) gives the program information on whether a beast is dead(0) or alive(-1 is true), if there are reserves of enemies, then regenerate a new one.

After the updating of the beast counter has been done, we have to look at Marvin. He has killed an enemy by colliding with it, so his energy is reduced by 48 points. A flag is set to let the main program know that there is some printing to be changed on the display screen, and to print the new information. Now the energy variable is checked on to make sure that it will not fall below a value of 0. This is done by using the MAX command. This is a really useful command and is worth mastering.

Energy=Max(0,Energy) This makes sure that the energy value will either equal 0 or the energy value if this is greater than 0, in effect, the energy value can never go into a minus figure.

Next comes the section of the procedure that deals with playing the sound effects. If the energy falls to zero, play a sample, if the energy value is greater than zero, play another sample, then use the Boom command to play with the effect of the enemy dying.

Procedure KLLSLIK[BNM] This procedure controls what happens if an enemy steps on a slick that Marvin has dropped. First of all the BSTDE flag is set to tell the main loop that an enemy has died. The enemy co-ordinates are grabbed using the Xbob and Ybob commands, and the bomb is shut down.

The next line decides if the dead beast will leave a coin for Marvin to collect. There are two random commands in this line - If Rnd(3)=1 Rnd(3)=2 - There is a reason for using two randoms, we needed to get a good balance between the amount of times that there will be a bonus coin left by a dead enemy and the amount that there would be no bonus.

When there was a single random, the coins rarely appeared. The easy answer would seem to be to reduce the number in brackets, which was 6 to a single random. We tried this, but the result was that a coin appeared just about every time an enemy died. After playing around trying different things the double random seemed to give the best balance for the coin. There is no special reason for this, it is just a case of 'it works, so use it!' The coin is to be planted on the map, then the variable NCOIN is increased by



Will this Beasty leave
me a coin?

The next bit is a tricky bit of TOME work which covers if there is more than one coin available to be picked up. The value of OX (dead beast number-1)/16 is put into the variable OXX(NCOIN). Put simply, this means that the enemy co-ordinates are broken down into tile co-ordinates.

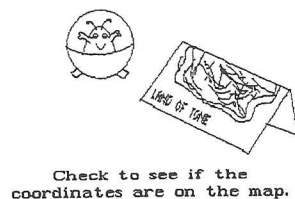
In Tome two sets of co-ordinates are needed - because Marvin is made up of 16*16 tiles, dividing the screen co-ordinates by 16 will give you the tile co-ordinates. A flag is then set to true telling the main program that a coin has been put down. This finishes the If End If loop with the randoms. Now we have to remove the slick from the map. First of all the flag is set to false ie SLIK=False, to tell the main program that a slick has been lifted.

Now we Map Plot 0, SLKX and SLKY. This plots a floortile(0) in memory in the co-ordinates that we grabbed when the slick was dropped. Now calculate the screen co-ordinates of where to put the floor tile and plot it on the visible screen.

This did not need to be put into an array as a second slick cannot be dropped until the first one has disappeared, thus resetting the timer. The above does not affect the screen display. The slick is displayed on a tile that has to be replaced by a floor tile in exactly the same screen co-ordinates. This is calculated with the following formula $TX = SLKX * 16 - (MX/16)$ which converts the tile co-ordinates to screen co-ordinates.

The same is done for the Y co-ordinates. The values now held in TX and TY make up the XY co-ordinates of the place to put the tile on the viewing screen. The next step is to check that the screen co-ordinates are legal, that is that the result of the calculation is actually on the screen.

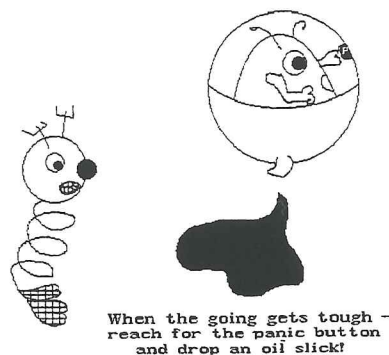
To do this, we check that TX is greater than or equal to 0 and that it's less than 272. If it is less than 0 or greater than 272, it means that it is not on the screen. The same check is done for TY, using the values of 0 and 192. The tile is only placed if the co-ordinates are inside these limits.



Next, the map is set to screen 1, the work screen, and the icon is placed using Paste Icon TX,TY,1. There seems to be a bit of a conflict here, said Paste Icon 1, and Plot Icon 0, this is because Tome counts from 0, while the Paste Icon command assumes from 1. This means that you have to add 1 for pasting icons and subtract one for Tome.

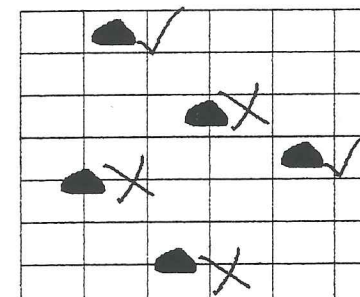
Now set the screen to screen 0 and end the If End If statement. SLKTIME is now set to 0, This variable is used to control the time that a slick is visible on the screen. Add 20 to the player's score and again set PRNT True so that the program knows that it has to update some information on the panel.

The rest of the procedure is identical to BSTDEAD where it checks to see if an enemy is alive etc. DUMPSLIK This procedure drops a slick onto the visible map as well as the memory map when the fire button is pressed. It must first find the tile co-ordinates on which to place the slick tile. The player's co-ordinates are used in conjunction with the map co-ordinates to work this out. These are two separate entities, but they must be added together for accuracy in detection or placement.



This is done with $SLX = (X + MX) / 16$. SLX is the variable used in Procedure KLSLK, and this will be equal to the co-ordinates of the grabbed tile. X is the player's co-ordinates on screen, and MX is the place he is on the map. Then the same is done for Y.

Map Plot 117, SLX,SLY comes next. This works out the XY co-ordinates of where to put tile 117, which is the tile with the slick. This is followed by working out the screen co-ordinates of the place to put the slick.



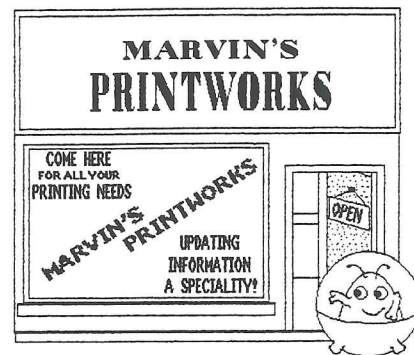
A slick tile must be placed correctly so that it does not overlap other tiles.

You could say, why not use $X + MX$? This would work, but only after a fashion. If your map is moving at a speed of 4, then the tile might be spot on target, but it could also be 4, 8, or 12 pixels out. The screen moves four pixels at a time, the values of TX and TY must be calculated again using the same formula as before.

Point to screen 1 and Paste the Icon on screen 1. The icon for the slick will now be $117 + 1$, for Paste Icon, ie 118. Point to Screen 0 and reduce the energy by 2 points. Set the flag PRNT to true so that the main program will update the panel, then check the energy level as we did before. The appropriate sound sample is played, and the procedure is ended. Procedure PRNTIT is the driver routine that updates all the information on the panel.

The last thing you want when writing a routine like this is to let the program run through this lengthy procedure during every pass of the loop. The way to stop this happening is to check that a condition is true before calling the routine. You will have seen 'PRNT=True' scattered through the listing in other procedures, what this statement is doing is allowing

Amos to access the PRNTIT procedure only if PRNT= True, if it is True, then the procedure is ignored.



Procedure PRNTIT is where all the information is updated and printed on the screen.

If the program were allowed to enter this print routine at every pass it would slow down the program and give sluggish gameplay. The separate print conditions are themselves all conditioned so that they only come into play when needed, but as there are so many of them, this would still take up much time. It is important to remember that even a jiffy, which is one fiftieth of a second, is a long time to a computer so every time saving method should be used.

The first thing in PRNTIT is to set up the graphics writing mode to allow to create a shadow effect when printing the text for the panel. The graphics writing mode is set to 0 which allows text to be printed over something that is already on the screen without the original being totally obliterated, i.e. the new text is printed on a transparent background. This allows us to print the text twice in two different colours, the second slightly offset from the first to give the shadow effect as explained later. Next, as we are working with Double Buffer, we need total control over the screen updating to make sure that there will be a rock steady display while the panel is being updated.

If this control is not maintained, the screen will flicker and the effect will be spoilt. This is done by turning Autoback to 0. Bob Clear is now used to force Amos into a condition where it is ready to update all bob positions

and bob masks. There are several things that are updated by this routine, in fact there are ten different things that the machine has to check on. These are Energy, Keys, Diamonds, Gold, Trumpet, Horn, Harp, Coin, Score and Lives.

Each one of these has two variables attached to it. Let us take Score as our example. We have the variable Score which holds the player's current score, and we also have OLDScore which holds the value of the player's score from the last time that the score was changed in this loop. To stop Amos working too hard and slowing things down if it updates all ten conditions every time that this routine is called, we do the following. We check to see if the current score is higher than the previous score, if it is, then it needs to be updated and printed, if it is the same as the old score then the condition is ignored and the program goes on to the next condition.

This is a method that suits the way we program, as everything that needs to be updated in this way can be put into the one procedure where it can be made to work properly and then shut down. If updating is needed, you only have to look at this procedure. Eight out of the ten conditions are dealt with in this way, only two have to be done differently. These are Energy and Keys, we'll deal with these later. The following explanation covers Diamonds, Gold, Trumpet, Horn, Harp, Coin, Score and Lives which are dealt with as follows:-

This time we'll use GOLD as our example. First of all the value held in GOLD has to be turned into a string using the formula `GOLD$=Str$(GOLD)` then there is an extension to the command which is not in the Amos manual which is `-"` " This little bit is a very useful facility in Amos which we didn't know about for a whole year, but now we use it frequently. Here it looks inside the target string, here it is `GOLD$` for any occurrence of the source string containing whatever is inside the sets of inverted commas. In this case it is a space. It then removes any such occurrence from the target string.

The reason this is used here is not really obvious, it is because when you turn a numeric variable into a string variable, the computer automatically saves out a space in front of the number. It does this because it will always assume that a number is a plus unless told differently, and so leaves a space in case that number should at some point become a minus. The space is taken out here as that extra space can cause chaos when displaying the

number, so it is better to remove it.

The next command is `Z$=String$("0",Z-Len(GOLD$))`. This little bit of a line is part of a very useful function. It is put in because of the way in which numbers are displayed in the game - even if Marvin has only collected one bag of gold, it is displayed as two digits 01.

The above command works out how many zeros to put in front of the digit. It does this using the `String$` command. This command will return into the amount of whatever is inside the inverted - in this case it is "0". It repeats itself `Z-Len(GOLD$)` times.

Example, when Marvin has nothing, the result of the transaction is just zeros as there are no other digits to be taken into account. If he has one bag of gold, the calculation works out that this is one digit, so there will need one zero to make the display show a two digit figure i.e. `Z$=0`. If he has 10 bags, then no zeros will be needed i.e. `Z$=0`. The next part is to add the result into `GOLD$` which we are going to print by using `GOLD$=Z$+GOLD$`. The transaction is now complete. This now gives us numbers in the string so we can print them.

This routine might not look worth the effort, but if, as with the score, you are working with six digits, the benefits become clear. If you print a number that's inside the variable, you will find that the numbers move one pixel or two from left to right, giving a wobble, the more digits used, the more noticeable it becomes. Your aim when writing a program should be to make everything that is visible appear as professional as possible, and this one touch that will help to achieve this.

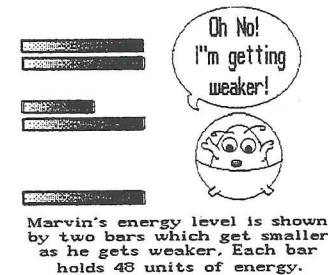
Now we have to clear a square where we want the value for the gold to be printed, this tidies up the display by rubbing out the old numbers. To create the shadow effect on the text we first choose Ink 29, which is black, and then using the `text` command, we draw in the value of `GOLD$` at the same position on the screen which is 79, 190.

Now using Ink 28, white, we draw the top layer of the effect, which is the same as the bottom or shadow layer, but this time placed one pixel up and one to the left of the first. You could think of this as planting a brush of white in DPaint over a darker brush of the same thing, displacing it by one pixel up and across from the original. Try it out in DPaint to see the effect.

magnified.

As we turned the Gr Writing mode to 0 at the start of the procedure, the two layers will be merged. The next step is very simple but nevertheless vital. `OLDGOLD` must now be set to the value of `GOLD` so that it will not be updated on the next call to the routine, unless needed, thus speeding things up. This explanation covers the updating of all the eight conditions listed earlier.

Now we'll explain the other two. The first is for the updating of the number of keys held. This is identical to the other conditions in the way it is handled, the only difference being that all four keys are updated even if just one number changes. This is done so that there will be no need for four separate "If End If" statements. Last, but not least comes the Energy factor. This is displayed as coloured bars which decrease in length as Marvin loses energy, this, we thought would look better than just having



it displayed as a decreasing percentage figure.

Because the energy is shown as two bars, the calculations have to be split into two sections. The first step is to check the new energy level against the old level to see if it has changed, if there is no change, the condition is ignored. Clear the space needed to display the new information, this also makes sure that you are not calculating on a value obtained after Marvin has died.

A check is made to see if the variable `ENERGY` is greater than 0, if it is, then Marvin is still alive and the two variables `TEN` and `TEN2` are set. `TEN` holds the value for the top bar and `TEN2` holds the value for the bottom bar. This is easy at first as you can assume that the bottom bar will not be

touched.

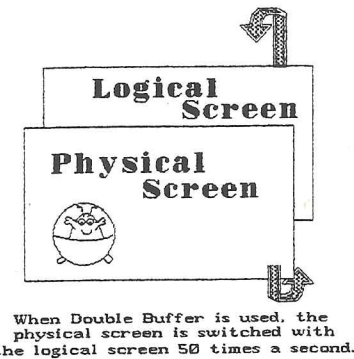
To find out how much of the top bar is left we say that $TEN=ENERGY$. Each bar is 48 units of energy long, and the energy level was set to 96 (ie 48) at initialisation, so as the energy level falls, you subtract the length the bottom bar (48) from the energy level to give you the length left on top bar. So, If $ENERGY=94$, Then $TEN=46$, so we need to draw the Energy bar 46 units long. Now to check on the second energy bar. If the energy level is less than 49, then the transaction is simple, $TEN2=ENERGY$.

The top bar has been used up, so we are now working on the bottom bar. The second bar is drawn first, the whole of the bar is drawn, then if there has been a loss of energy, the bar is redrawn to its new size. If the first bar is still active, ie TEN is greater than zero, then the top bar is drawn. In real time each transaction draws three bars, each one being a different shade of green, lying parallel to each other, this gives the 'tube' effect which is much better than a bar of flat colour, it's another touch to give polish to the program.

Once the program has read through all the If End If conditions, and acted on if necessary, there are only a few things left for the procedure to do. The one in front is known as the Physical screen, when the screens are switched, the Physical becomes the Logical screen and vice versa. All this is done automatically. Double Buffering is vital for all programs where you have bobs moving - there is no substitute. Autoback 0 completely disables this screen swapping so that everything is placed on the Logical screen while the Physical screen is left untouched.

First of all PRNT is reset to False so that it won't come back here until told to do so when something needs to be updated. The bobs are redrawn as an alternative? Autoback 2 is the default setting for Amos, and works very well with Bob Draw. The panel has to be copied to show the new display, this is done in two slabs because it is an irregular shape, and needs two screen copies. Why is there a need to do things manually if Amos gives you an automatic alternative? Autoback 2 is the default setting for Amos, and works very well with Bob Draw. The panel has to be copied to show the new display, this is done in two slabs because it is an irregular shape, and needs two screen copies.

At the beginning, Autoback was turned to 0, which means that all transactions in the above conditions would have been done on the Logical Screen. In case you are unfamiliar with Double Buffer screens, the following may help you to understand the terms used. When using the Double Buffer command, Amos creates an exact duplicate of the current screen behind the visible one, like two sheets of paper laid one on top of the other. What Amos does is that when something happens on the screen, it actually happens on the Logical screen, which is the one you cannot see. Every 1/50th of a second, Amos swaps the two screens.



In quite a lot of cases this is fine, but because we use Cls, even though just to clear a little square, and also text, both of which are relatively slow to transact, put with the ten "If End If" conditions to be read through this puts the process over this time limit which made the display occasionally flicker. This only happened once in a while, but if you want to produce a professional looking program, then all these things must be eliminated.

You should never say, 'It'll do!'. It won't! So the best decision was to do the screen swapping manually. This means that after printing the text updates, the screens must be told to switch using the Screen Copy command.

This time though, you must not use Screen Copy(0) as the screen number,

but Screen Copy Logic(0) to Physic(0), which is the destination screen. When this is done, the screen swapping is put back onto 'autopilot' via Autoback 2. This ends the print updating procedure.

The next procedure, CHKTILE, keeps a record of the tile that Marvin is standing on, has he walked over some gold, a key or a trumpet? It also checks for doors and the important extra life symbol if it exists. The first line of the procedure does a bit of garbage collection, ie it brings back some of the memory that has leaked away during the program. This line is PIPPA=Free + Chip Free + Fast Free.

This procedure is called by another procedure, ProcCHKWALLBU, where we find the value of MT. MT will equal the tile number of the Marvin is standing on, this is the Icon number. If Marvin has stood on an extra life tile, add one to the number of lives and make sure that the flag tells the main loop that this extra life has been put on the map is set to false.

Now comes the familiar PRNT=True to get the information updated on the panel. As we do not want Marvin to be killed by his own slicks, we check to see if the tile he is standing on is a slick, if it is, ignore the fact and turn the slick off. The keys play an important part in the game, you need keys to open doors, and to make things more interesting, they come in four colours which will only open doors of the same colour.

To do this each key colour is given a variable, so that a record can be kept of each colour, eg if a blue key is found, then one is added to the variable BKEY. A variable is now set to hold the value of the tile that displays the score for picking up an object, 25 is added to the score and PRNT is set to True.

Most of the other items in this procedure are dealt with in a similar manner to the above, the only difference being that as the compulsory items are collected, ie the ones displayed at the bottom of the screen which you need to get onto the next level, the score is subtracted from the start figure. Here the numbers refer to the amount left to get rather than the amount already found. Again the Max command is used to make sure that the level does not fall below zero.

There are more of these items displayed than are needed as otherwise the game could be impossible to complete. The map area is very large, and would

look empty with just the important items mentioned above scattered around. If we had increased the number of these to fill up the map, then the game would become too easy, so we put in other objects that would give points, but do not have any other purpose. These were dealt with in a different way to the other objects.

The score value for these is set on a random ranging from 15 to 1000. The higher values come up much less frequently than the lower ones. A condition is set for each object that is picked up, and they are dealt with as before. Now we deal with the task of placing the tile that gives the bonus life somewhere on the map. This is done by Procedure PTLIFE.

To get this extra life, the player must have found enough coins on the previous levels. The location of this life tile is calculated by the lines:-

```
RDX = Rnd (Map X-1)
RDY = Rnd (Map Y-1)
```

Map X and Map Y are Tome commands which give you the number of tiles on the X or the Y axis. Using a random on these figures -1, will give a random tile number on the X axis and another on the Y axis which is the tile number where the life tile will be placed. eg it could be the 30th across on X, and the 18th on Y. We generate a label PTK9: If the tile is not a floor tile, then the program loops around to this label and tries again. Another Tome command is used next, this is Map Tile. The line is:-

```
RT = Map Tile (RDX,RDY)
```

This gives us the value for the co-ordinates at RDX, RDY which is the place that the life tile will be placed. Now we have two If End If conditions. The first covers the coordinates being acceptable, that is if they refer to a floor tile. If RT = 0, (a floor tile) then Map Plot 116 (the bonus life tile) at the coordinates RDX, RDY.

If the random has failed to find a floor tile, ie if RT is lesser or greater than 0, the floor tile, then add 1 to the value of RDX, check to see that RDX does not go beyond the map limits, if it does, then it is reset to 1, which puts it back to 1 on X. RDY is done in the same way. Then we go back to the label PTK9 to try again.

This carries on until the coordinates are acceptable. It may sound as if it could make the program hang, that is stop, until the right co-ordinates are found, but during time trials done when writing this routine, it was found that it rarely looped around more than three times.

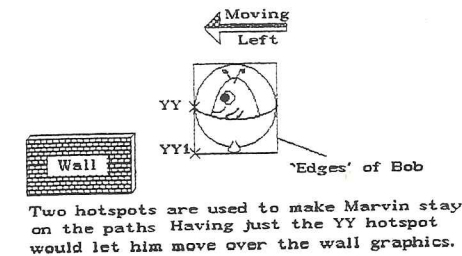
As this is a maze game, we have to make sure that Marvin stays on a straight and narrow, and does not cheat by walking over walls. This is done in the procedure named CHKWALLBUMP.

This procedure is mentioned in Proc CHKTILE, as it is a part of the procedure. Firstly we check on the value of DX and DY, these variables hold information about the direction in which the joystick was last pushed. They could only hold two values each that matter, $DX = 1$, means that the joystick has been pushed to the right, or $DX = -1$ that it has been pushed to the left. Similarly with DY, a value of 1 means the joystick has been moved down while -1 means up. It is important that these variables are stored as you will need to know several times which way the player is going.

Here they are used to find the tile coordinates of the player's position on the map. This means calculating on X, which is the position of the player on the screen, and MX which is his map position. Both of these coordinates are needed in calculations for games written with Tome. Now we attempt to find the values of XX, YY, XX1, and YY1. The reason for needing two sets of coordinates will be explained as we go on. We'll take the following line as an example.

```
If DX=-1
Then XX=(MX+X)/16:
YY=(MY+Y)/16:XX1=(MX+X)/16:
YY1=(MY+Y+8)/16
```

After this calculation, XX will hold the the X map tile coordinate of the hotspot of the player in tile numbers. YY will hold the Y map tile coordinates of the hotspot of the player. XX1 will equal the map tile coordinates, as does XX, and YY1 will hold the Y map tile coordinates of the player's hotspot plus 8 pixels. The other DX and both DY's are handled in the same way using an offset of plus 8 pixels. The reason for this offset factor can best be explained with the help of diagrams.



A bob, whatever it's apparent shape, is really a rectangle. The bob containing Marvin must be allowed to move along the paths of the maze, but not on any tiles that are parts of the walls.

To do this a point on the bob is set as a hotspot, ie a point that can be used to detect when Marvin bumps into something else. If we set a single hotspot, it would be possible for Marvin to walk over parts of the walls.

By setting a second hotspot, on one of the corners of the bob, he will be stopped from walking on the walls. The corner chosen by the offset of 8 pixels depends on the direction that the player is moving. The calculations mean that Marvin will only be able to walk along paths of whole tiles, and never be able to move across half a tile. The setting of two hotspots still does not give 100% coverage, but as it is here linked to the direction in which Marvin is going, it is sufficient.

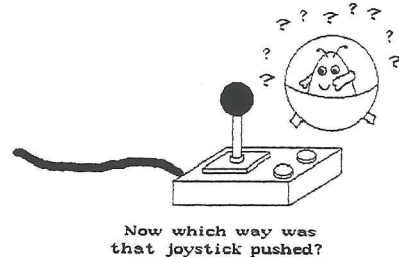
A third hotspot at the other corner on the same side would give full coverage. Tile Val is a Tome command that holds the value of a tile in the tile list. After these calculations, TV will hold the Tile Val of the tile at XX,YY,0. This is $TV = \text{TileVal}(XX,YY,0)$. The zero means to look at list 0, and respectively TV1 holds the value for XX1 and YY1. Armed with this information we can check on a few things.

The first thing - is the move legal? That is did he walk into a door or a wall? To answer this we use an If End If condition, If $TV > 0$ or $TV1 > 0$ and If $TV = 250$ or $TV1 = 250$ then reset MX,MY,X and Y to their old values

which were stored earlier - this in effect stops him moving. Every time the joystick is pushed to the left four is added to the map and player coordinates, as yet there is no visible result on the screen. A check is done to see if the move is legal, if so, update as normal, if it is an illegal move, then put the player back to the coordinates he was on before the stick was moved.

Now we get the value of the variables used in CHKTILE with MapTile(XX,XY). If End If is used again in If MT>79 and MT<200 CHKTILE : End If. This means that if the value in MT equals no less than 80 and no more than 200, it means that Marvin has walked over something important, so we'd better go and check on it. The three important variables are reset, MT=0, V=0, and TV1=0. This ends this part of the program.

In Procedure CHKPARAM we check on two things - the allowed parameters of the map, and the player. Briefly, this procedure checks to make sure that you cannot run off the edges of the map and that the player stays



roughly in the centre of the viewing area as the map scrolls, unless edge of the map world is visible, when he will be allowed to move over to limit of the map.

Procedure SETUP, as the name suggests, sets up the map and player coordinates for initialisation. The speed of the player (Marvin) and enemies is set to 4, ie they will move four pixels at a time, and Bob Update is turned off. In this procedure the variable HXO is set to true (-1). This is used as a flag to initialise the map drawing routine which draws the whole map used in the NICEMAP procedure.

Procedure NICEMAP is mostly taken from a routine written by Aaron Fothergill, the author of TOME. It is supplied with the program for you to use to scroll the map. The first thing it does is to point to Screen 1 which is the work screen where all the moving work will be done. There is a good reason for having two screens opened here, one to do the work and not visible to the player, and the second which the player sees. The viewing screen is Double Buffered as it will have Bobs, and it would take too long to do the map on a Double Buffered screen, so we have an unbuffered work screen where the construction work takes place.

This is then screen copied to the viewing screen when the work has been completed. This is a much faster method and is a must for any Tome work. Now we check on the HXO variable which we set in Procedure SETUP.

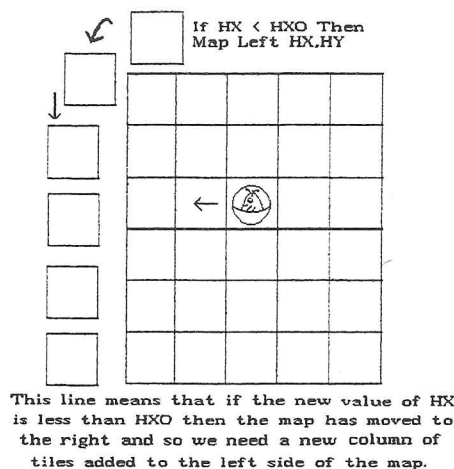
If this is set to true, then we carry out a Map DoHX,HY command, which draws the whole map starting from HX,HY using the parameters set at initialisation. Also if HXO =True, skip the rest of the procedure and carry on with the rest of the program. This procedure is called at the very beginning of the program and also at the start of every level to initialise the map display. The rest of the procedure is only transacted if HXO is valued at anything other than -1. The next few lines come from Aaron's routine which we mentioned earlier.

Unfortunately we cannot break these lines down to explain how they work, as we do not fully understand them! Here we have gone on the principle, if it works, use it! The routine is a general map driver and it copies the required part of the work screen to the viewing screen as it is needed. For example, if the joystick is pushed UP, the map itself will move DOWN, so the program needs to copy a new slice of tiles to go at the top of the viewing screen.

The same will apply for movement in the other three directions. The routine is the main driver that makes the screen move in jumps of less than 16 pixels, which is the size of the tiles used here, this would result in very jerky scrolling. The variable HXO is not only used as a flag to initialise the map for the rest of the game, but is also used to keep track of where the screen WAS before it needed to scroll, therefore it is a very important variable.

The four conditions that follow are vital to make the map scroll in the desired direction. We'll explain them by using one as an example, but the principle is the same for all four. HX controls the scroll from left to right, HY controls the scroll from top to bottom

If HX<HXO Then Map Left HX, HY



In English this means, if the new screen coordinates are less than the old screen co-ordinates on the X axis, then a new piece of the map is needed on the left side of the viewing screen, so put in a new column of tiles.

The same applies for the other three directions. This is a small routine, but it is the nucleus of the method for moving the map around.

Procedure INIT2 comes next. This procedure, unlike ProcINIT, initializes the program in-between levels, and covers quite a few conditional events. We'll take them in turn. First it erases banks 6 and 8 which are used for TOME, Bank 6 for the map data, and Bank 8 for the Tile Val data. Next the information containing which level the player is on is put into a string, even if it is level 1 at the start of the game. Again we use the "-" command to remove the extra space saved with the string, as we did in Procedure PRNT. The reason for this will be given later.

The game is level structured into three sections, levels 1-5, 6-10 and 11-15, so we next check to see which graphics are to be loaded in. This is done quite simply, take levels 1-5 as an example.

Check to see which level you are on with If LVL>0 and LVL<6 ie if the level number is between 1 and 5 inclusive, then that's the set of graphics needed. Now we set the number of beasts for the levels which for the first set of levels is 30. Then the data for the enemies in levels 1-5 is loaded in and stored in Bank 4.

Now we come to the tiles and bobs for the levels. All the tiles needed for each set of levels are stored in one file as are all the bobs for each set of levels, because of this we do not want the program to reload the same file when passing from one level to another inside the same set, this would be a waste of time, and unnecessary disk access should be avoided.

The bobs for each level are different, but they come out of the same file. To see if the file has been loaded, we use the following:- If Length(2)=0 - ie if Bank 2 is empty, load in the tiles file for this set of levels. If Length(1)=0 - as above load in the bobs file. Therefore if the length of the bank is more than 0, ie it has something in it, then do not load anything!

When a set of levels has been completed, the two banks are erased so that the program will be forced to load a new set of graphics. This ends this condition.

For the later sets of levels, we check as before, for levels 6-10 the number of beasts in the level is 35, and for levels 11-15 the number is 40. The tiles and bobs are loaded as before by checking on the level number.

This number of enemies in a level is the number of 'lives' given to each enemy, as previously described, before the amount of enemies appearing starts depleting.

After all this, the command No Icon Mask is used. We do not need an icon mask as there are no transparent colours used in the icons, ie you cannot see the background through the image. Creating icon masks uses a lot of memory and should be avoided wherever possible.

Unlike the bobs and the tiles, the map layout is different for each level and new one will have to be loaded as each is completed. These are small files and will be very quick to load anyway.

Loading files for a multi-level game is an area of programming where many programmers create long-winded routines that could have been done very simply without using up loads of precious memory with data statements.

We came up with the following method of loading files, which is a lot smaller than the text used to explain it to you!

What we need to do is if the level number is 1, then load the data for level 1 map.

This is how we tackled it. First of all we created a directory to store all the level data with the file names as follows, lvl1.map, lvl2.map etc. 'Lvl' means level, the number following is the important level number, .map is just to say to anyone looking at it that it is map data rather than bobs for example.

This is where the first line of the procedure comes into effect, remembering turning the level number into a string? We create the file names by building up a string using

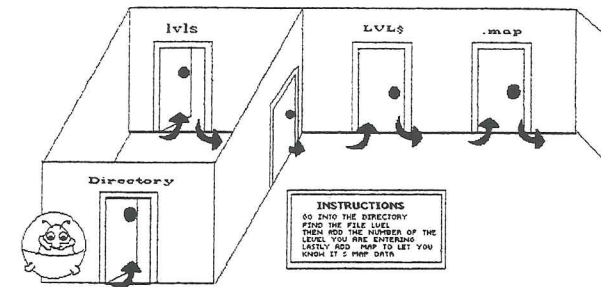
```
MP$="lvls/lvl"+LVL$+".MAP"
```

This builds up the correct file name to load for each level as follows.

"lvls/lvl" tells the Amiga that the file is in directory lvls and that the beginning of the file name is lvl. (The / character signifies a directory.) Added to the string the string value of the level obtained earlier, with the space removed, it was important to remove this space because otherwise it would come out as lvls/lvl 1.map which would cause a 'file not found' error message to appear as the file name on disk does not have a space in it. The string is finished off by adding ".map".

So as the value of the level changes, so does the file name to be loaded.

It is good practice to structure the file names for the levels in a game in



Loading the data for the different levels of a game is easy if you build up a string that tells the Amiga what to load.

logical manner so that the complexity of the programming needed to load them in can be reduced.

The data for Tome's maps is loaded into memory, so we need to know the length of the file. The memory hungry way of doing this is to store the information as data statements and read them in as needed. By now you should know that we would not choose this method! We chose an easier method involving less coding and less memory.

```
Open in 1,MP$
```

This opens a channel to the disk drive and looks at the file in MP\$.

```
FL=Lof(1) Close 1
```

This gets the length of the file, then the channel is closed telling the disk drive that you have finished with it. It is good practice to close the channel because if left open, it may cause problems later in the program.

Now we have to find out how many bobs are in the bank. This is needed as the amount of bobs for each section varies considerably, so this is one place where memory saving is more difficult. There is one constant factor in these bobs and this is the explosion effect which is needed all through the game. Rather than storing the same set of bobs repeatedly with each set of level bobs, we used the handy facility of being able to append these bobs to an existing bob bank. The main problem here was finding out where the explosion bobs were to be added as each bob total is different. The solution was not too difficult, we created a variable NBIB (number of bobs in bank!)

as an offset variable.

In this we store the number of images in the main bank before the explosion effect. The amount of bobs is found by using `NBIB=Length(1)`. With `ico` and bobs, `Length()` gives you the length of the bank, but here it will give you the amount of images in the bank.

Now we are armed with the number of bobs in the first bank, we can safely load in the second set. This is done with `Load"Boom.abk",1` the figure 1 is the important factor here, a normal load does not need this. If the number is left out, the first bank will be erased, using "1" will append the new bobs to those in bank 1.

Now we must prepare the machine for the map data. As pointed out earlier, the data is loaded into memory, and unlike other types of loads, you have to prepare a place in memory and reserve a bank. `Bload (Binary load)` will load information from disk exactly as it finds it. First of all we have to reserve the necessary memory with:-

```
Reserve As Work 6,FL
```

This reserves bank 6 as work space, FL bytes long. FL was obtained earlier with the `Lof` command.

To load something, you also have to put in the start location.

```
Bload MP$,Start (6)
```

`Start()` gives you the start position in memory of the bank number inside the brackets, assuming that the bank exists.

This is easy to test inside Amos' Direct Mode. Enter Direct Mode, then key in:-

```
Reserve As Work 5,5000 <<Press Return>> Print Start (5) <<Press  
Return>>
```

This will give you a seven digit number. There is no point giving a specific example here, as the result will vary between machine models etc.

To move away from Marvin for a moment, there are several forms of `Reserve As` which it may help you to understand so that you can completely control the total use of memory that Amos has been given by Francois Lionet. This control is over Fast and Chip memory.

You can `Reserve As Work` or `Reserve As Data`. At first there doesn't seem to be that much difference, but they do have different uses.

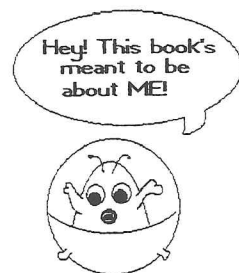
When you are developing a program inside Amos, using `Reserve As Work` means that all the data stored in these banks will be ignored when you save out your program. This is handy for trying out things that you know are temporary. On the other hand if you use `Reserve As Data`, the data inside the banks will be saved along with the listing automatically. This means that you do not need to put these banks on disk again and load them in as separate banks from inside the program. They will already be there in memory ready to be used!

This is something we often see when putting the Amos PD Disks in order, there is no need to waste disk space and access time if you have used `Reserve As Data`.

To test this, if you are nervous about losing data, backup your program, then erase the banks you reserved as data leaving in the load commands. The program will still run.

It can also be used to reserve memory in either Fast or Chip. Using `Reserve As Work` or `Reserve As Data`, you do not control where the data will be placed, if Fast is available, it will use that first, if not then Chip memory is used. If you want to make something load into Chip, you can use `Reserve As Chip Work`. One useful application of this is for loading in sound samples as the Amiga will only work on sounds stored in Chip memory if you use the `Sam Raw` command.

If it has not been forced into Chip, it will be heard only as a series of crackles, in Chip it will play as intended, this is a useful tip to remember. Marvin was written for 1mb, so we knew that Fast memory would be available, so chip memory was not forced.



Now we use Reserve As Work(8),1024 - 1024 being the length of the Tile V data. This data is always 1024 bytes long, so no fancy method of checking needed here! The size of a map data file will vary according to the size of the map. If you look at the directory lvl, you will see that the file sizes range from 2504 in the lower levels to 10,004 in the higher levels.

Blod this into Start(8), then jump to BSTDTA which will be described later.

The next few lines took a lot of Getting and Putting of blocks, Cls's and Pasting of icons. This is like a construction site building up a display on the Logical Screen 0. The program is putting together the display of the collectable objects, etc also it turns the gr writing mode to 0 to allow the shadowed text effect as described before. When all the graphical alterations have been made, the Logical screen is copied to the Physical screen and we go to PRNTIT to finalise the initialisation of the display. If you could halt the program here and take a look, you would see the playing screen with all the panel information, but as yet no map.

The map will be displayed when we go to the main loop of the program. There is no special reason for it not being displayed here, it was just a matter of personal choice.

Procedure INIT is used on only two occasions, once on initialisation, and then only on the death of the player. It is important when a player dies as you must remember to reset or initialise all the relevant variables used at the point to which the player is returned.

Now we Erase Banks 3, 2 and 1. All the work screen variables are reset, the work screen is deeper than the viewing screen to stop juddering. Screen 1 is opened again as a way of cleaning off all unwanted displays. Load in the samples from Bank 5, if the Length of this bank is 0, by now you should know that this means that the bank is empty. If the length is anything greater than 0, then the samples are already in memory, so there is no need to load anything. Bank 5 is always used for samples.

Tile Size 16,16 is a Tome command that tells the program that tiles measuring 16 x 16 pixels have been used. Another Tome command is now used, Map View 0,0 To WKX,WKY. this tells Tome how much of the map to display when transacting the Map Do command. For example - if we have Map View 0,0 To 100,100 the map will be displayed in a rectangle which extends from screen coordinates 0,0 to 100,100. This can be enlarged or reduced.

We next load the graphics for the control panel into Bank 15, it is Unpacked to Screen 0 and Bank 15 is erased. Screen 0 is displayed off the viewing screen so that it is hidden. A lot of the graphics and sounds could have been left in memory, but by now, memory was getting tight, as Marvin had now grown from a small idea to a rather large game!

The screen now has to be Double Buffered to prepare it for bobs. This is another situation that has caused programmers trouble. It is wrong to assume that once a screen has been Double Buffered, it will remain Double Buffered - WRONG!

The Double Buffering is lost when you load in an IFF screen or when you unpack to a screen as both these processes shut down the screen, then reopen it. As has been mentioned before, this is the only way that Double Buffer can be turned off. So remember to redo the command if you have to unpack or load an IFF file to a Double Buffered screen.

It is a pity that this is the only way to shut down Double Buffer. The only way that you can turn it off and retain the same display is to create a screen of the same size behind the Double Buffered screen, get the palette of the first screen, then screen copy to the first screen which will shut off Double Buffer.

Procedure STRTLE[a,b,c] is a very small procedure whose only purpose is to

keep a track of how many bonus numbers have been placed on the screen. When Marvin walks over a collectable object, a number is displayed for a short while to show the number of bonus points. This procedure allows up to ten bonus numbers to be displayed at the same time. Without this routine, the first number would disappear as soon as Marvin walked over a second object. There are many places in the game where bonus objects are placed next to each other, so you will be able to see this routine in action.

Procedure PTTILE is the opposite of the one above which logs and stores the tiles as they are revealed, this one removes the tiles.

It works in a For Next loop, the variables are stored in a Dimmed array and are extracted one by one, and a floor tile is placed at the coordinates obtained.

HOW A TILE IS PUT DOWN

This is very similar to the other routine, it puts a specific tile at a specific place on the map. It not only puts it on the visible map, but also in the map data for the screen.

This method is, if done carefully, very simple to use. First of all use the Map Plot command to plot the new information into Bank 6. Then if the tile is still visible on the viewing screen, which is not always the case if the player has moved away, paste an icon at the relevant place on screen 1, which is the work screen. No more is needed as the general screen updating done every 1/50th of a second in the main loop will do the rest for you.

Now we come to the procedure that controls all the enemies, whether they are on the viewing screen or out of sight. The procedure is named ENEMYBLOKE, and is quite lengthy. Do not worry if you cannot understand how it works when you first look at it, it's not as complicated as it looks when it is broken down into smaller sections.

There are a lot of 'ifs and buts' in this procedure as there are so many conditions to cover. We first of all set a variable to hold the information for the direction in which an enemy is moving. This enables the program to change the direction if it should bump into a wall.

It is not good practice to use lots of randoms inside loops, so we used the

method. Rather than put this into a loop, we create the variable at the start of the procedure so that it is only carried out once.

The direction is decided by using $WADY = \text{Rnd}(3) + 1$. The enemy can move in one of four directions which are numbered from 1 to 4. 1 = up, 2 = right, 3 = down, 4 = down. So why then do we use a random of 3 plus 1?

If we had used $WADY = \text{Rnd}(4)$, we could be left with a zero as a random number which would cause chaos in the program. Giving a random of 3, then adding 1, makes certain that the number obtained will never be more than 4, or less than 1.

A flag is created next, $DIS = 16 * 6$. This controls the limits of the map inside which the enemy bobs will be updated. This area is the whole of the visible screen plus a border of six tiles all around this visible area. The enemies are updated and moved if they are within these limits. If they fall outside the limits, they stop moving and stay at the point at which they left the screen, or at the place they were at initialisation. The limits of this system is that all the enemies outside this range will be static, but this is not a great problem.

Although we have a For Next loop of 40 which controls the enemies, it will probably only ever need to update about 12 of them as the rest will not be in action at the same time. By limiting the updating of the enemies to a certain area of the map, the gameplay is speeded up considerably. There is really no point checking on enemies that will not be activated before they are nearer to the viewing screen. The six tile border makes sure that any enemies just outside the viewing area are 'awake' and ready for action if the player moves in their direction. It also means that an enemy can move off the viewing screen in one place and wander back at another point.

Next we set some more variables which calculate the map screen coordinates by either adding or subtracting the value of DIS.

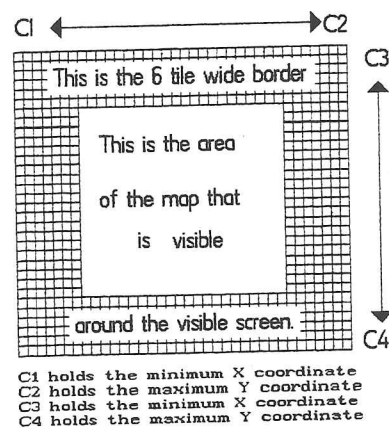
There is a For Next loop - For T=1 To BSTINLVL. Then comes an If End If condition. The first is If MORT(T), this holds the information on whether an enemy is alive(true) or dead(false). If MORT(T) is true, then the rest of the For Next loop is skipped, and the program goes to the next T.

The next condition relies on four other conditions being True. They involve

the variables C1, C2, C3, and C4, also OX(T) and OY(T). OX(T) holds the coordinates of the enemy on the X axis, and OY(T) Holds the coordinates on the Y axis.

C1 holds the minimum value of the X axis where an enemy could be, C2 the maximum on the X axis. C3 and C4 do the same on the Y axis.

If the enemy is outside these coordinates, then it is left alone, if it is inside them, then calculate on the enemies' movements. This is done with the



following line:-

```
If OX(T) >= C1 and OX(T) <= C2 and OY(T) >= C3 and OY(T) <= C4
```

If all these conditions are true then we are taken to the subroutine called UPDATE, if any are false, then Bob Off T=1. The '+1' is an offset as Marvin is Bob 1, and he must not be switched off!

The two End If's finish the If End If conditions and also the end of this loop.

Pop Proc is a way of instantly exiting from a procedure, but as we have said before, due to problems that have come to light with this command, it would be better to use a Goto with a label placed on the line immediately before the End Proc, which will do the same thing.

The first part of the subroutine UPDATE was called from inside the loop. A

counter is started which is set from 1 to 20. This is incremented every time the subroutine is called which is when it spots that an enemy is legally visible.

To add a bit of variety to the movements of the enemies, use the following line - If RD(COUNT)=2 : WAY(T)=WADY : End IF

This makes the enemies change direction every now and then when they haven't bumped into a wall, it stops them from walking around in a square as would otherwise do in some parts of the maze.

We now come to an On Gosub condition.

On WAY(T) Gosub WAY1A, WAY2A, WAY3A, WAY4A

The variable WAY(T) holds the direction in which the current enemy is travelling, so if it is moving up the map, the program will go the subroutine WAY1A to carry out the movement.

Next we increment the image numbers of the enemies, in other words, we animate them. This is done by creating two dimmed arrays. The first holds the maximum image numbers of the animated enemies. The second holds the image numbers in the Bob Bank. I(T) holds the value of where he is in his animation frames, NI(T) holds the maximum number of frames eg If an enemy has three frames of animation, then NI(T)=3.

Another memory saving feature is used next. This is the facility in AMOS 1.3 or later that allows you to flip an image on its axis. This means that you will need less bobs as the same image can be used to turn a bob around to face the opposite direction.

All the bobs for Marvin were drawn facing left, by making them all face the same way to start you will avoid confusion later on.

We now have to check on the way that the Bob is facing. Logic will tell us that if a bob is moving in direction 4, ie WAY(T)=4, then it is going left, so the image will be facing the right way. However if WAY(T)=2, then he will be travelling to the right and so to stop him walking backwards, we need to flip the image.

This is a very simple process which is possible with AMOS 1.3 or higher. (If you have an older version than this, get it updated!!) This is how it is done.

By adding \$8000 to the image number, it will be flipped automatically on the X axis, so that left becomes right. There are two other useful numbers, \$A000 will flip an image on the Y axis, and \$C000 will flip on the X and Y axis. Try adding these to an image number in your programs and see what happens.

Now we come to the part that checks on whether an enemy is allowed to be moved outside the visible screen, but inside the extra 6 tile wide border. If the enemy is inside this border, then the bob is turned off and we jump to the label KJLA: which is four lines away.

Now we position the bob with all the new information, the new X,Y coordinates and the new image number. The last thing we do is check to see if the next tile he will move onto will be a slick. If it is, then we go to Procedure KLLSLK and it's game over for this enemy.

We now check to see if the way in which the enemy is travelling is OK, or has he bumped into a wall? If he has, then add or subtract the value added or taken away from the X or Y axis to put him back where he was before he collided with the wall. All this does in effect is stop him going farther in that direction, then make WAY(T)=WADY, which will turn him around, but do not recalculate the move before going on to the next enemy. This gives a pause as if he has had to think in which direction he wants to go before he actually moves off. This pause is very short but without it, you would not get the effect of walking up a wall and having the necessary time to physically turn around to move in the new direction.

The next procedure is the one that in its original form stopped Marvin the Martian making an earlier appearance. It allowed the program to be compiled to speed up and smooth out the action, but crashed after only about a minute or two inside the game. Poor old Marvin was sent all over the place to try and find out what ailed him, but he stumped all the top Amos doctors and the cure was discovered by chance. We realised that only one or two of our programs would compile, whereas we had very little trouble with other people's work. After analysing our uncompileable programs, we found that they all contained the same random number generating routine. This routine was called repeatedly inside a loop which the compiler refused to consider. The original

routine created a loop which in turn created a list of numbers in a dimmed array, this list contained no repeats and no occurrences of 0. This sort of number list was needed in several areas of this game.

The new routine you see here is far quicker than the original. Although it is not always a true random, it gives a good enough result for the purpose for which it is used. With the old routine you could not randomize more than 90 numbers as it was so slow. The new version has been tried with up to 32,000 (yes, thirty two thousand) numbers as a test and came through it without a guru.

There is an Amos extension which has been written by Aaron Fothergill called Shuffle. This is on the Tome Goodies disk and is a true random generator which gives a better number shuffle than our routine. However both can be used to create your own interesting screen effects.

The routine itself is quite simple. First of all the Timer is reset to 0, then we wait a Rnd(5)+1 to allow the timer to move a bit away from 0, this means that the beasts will not appear at the same point on the map every time the game is run. Randomize Timer is used to create a new seed for the random (please see Amos manual for full explanation of Random) Next a For Next loop is created from 1 to the amount in Square brackets '[' after the procedure name, here it is AMNT. We now use a Repeat with another Repeat inside it.

```
Repeat :SWP=Rnd(AMNT): Until SWP>0
```

Now we come to another nice Amos command, Swap. This, as the name suggests, swaps variables or Banks.

We now Swap RD(SWP),RD(AMNT). Let's try to put that into English. It moves the last item in the list to a random place inside the amount of items in RD. This carries on for the amount of items in the list. Use the counter until they have all been done, then get out of the routine.

The first use of the old routine was used in one of our earliest programs, Thingamajig. Here it was looking for Rnd(50 to 90) and took quite a while to work, but since it has been updated to the new routine the time taken is negligible.

Procedure NEWBST[LK] is a relatively simple procedure that creates a new enemy after one has died, assuming that the limit for the number of enemies for that level (BSTINLVL) has not been reached.

We first find a random place on the map where the new enemy will appear. This is calculated with $RDX = \text{Rnd}(\text{Map X}-1)$, the same is done for the Y coordinate with $RDY = \text{Rnd}(\text{Map Y}-1)$.

The label NW2: is created to give a point to which we can return if the resulting tile is a wall tile or a bonus tile. The beast will be allowed to regenerate on the same tile as Marvin, adding an unavoidable hazard to the game play. This is done by getting the value of the tile on which the enemy is to be placed with $RT = \text{Map Tile}(RDX, RDY)$. If the result obtained gives the number of a tile other than 0, which is a floor tile, then try again as an enemy is not allowed to appear on anything else. If the result is tile 0, then the coordinates of the tile are stored in $ODX()$ and $ODY()$, not forgetting to multiply the number by 16, which is the size of each tile, and adding 8 to this to place the enemy bob in the centre of the tile.

If $RT < 0$ we do not go all the way back to the random as this would slow things down, we just add 1 to each of the previous numbers, so that the program will check the tile that is one across and one down from the original. As there are more floor tiles than any other type, it will not take long to find a suitable place to place an enemy.

When the floor tile has been found, loop back to the label. Now you have to make sure that you are still inside the map limits. We have done this with the condition `If RDX > Map X-1: RDX=1: End If`. This ensures a value of 1 is given to RDX if the legal limits of the map have been breached. This finishes off this procedure.

To place the enemies at their start positions on the map at initialisation, we go to Procedure BSTDTA. This works off the data stored in Bank 4. This data contains the initial direction of the enemies, their image numbers and the image limits. These are read into the respective variables ready for use when needed by using Peek. The reason we have used a data bank with Read here is that there is so much data, and this way saves memory.

The enemies are placed on the map in the same way as those which replace a dead enemy, with just a slight difference. NO bobs are actually put onto the

screen, but the data is stored in OX and OY ready to be used by the procedure OTHERBLOKE.

All this being completed, Bank 4 is erased from memory.

The next procedure BNUMBERS stores data in a very specific way using labels as line numbers in the data statements. This data contains all the bonus data, ie how many of each object must be collected, the limit of enemies and also the passwords for each level.

This is a part of the program that you can customise by changing the passwords, or cheat by making a note of the existing passwords! It is a way of having a look at the later levels without spending the time playing the game.

Labels are used as line numbers so that it is easy for the program to jump to the right line for each level's data.

This is done by using `Restore 1000+LVL`. This means that for level one, the value in LVL, here it is 1 will be added to 1000, so that line 1001 is the one read.

This method of data storage is very useful for games that have to store different data for each level, and also for adventure games where each room or location can be given a number, as the levels have in Marvin, so that the lines can be jumped to in the same way. This is a very fast and efficient way of storing and retrieving data.

Once the correct line has been found and read, the data is stored in the corresponding variables. These have been given very descriptive names: DIA, GOLD, TRUMP, HORN, HARP, BSTLMT and PWORD\$. Please note that although the actual objects change from level to level, the variable names stay the same, eg TRUMP will refer to trumpets on level 1, it could refer to a mushroom on a later level.

To make a screen slide on smoothly after a call to `Proc SCRNOFF[]`, we use `Proc SCRNON[]`. This is a very useful routine which you will be able to cut out and use in your own programs. To explain it simply, it increases the screen display on the Y axis until the screen is in position.

It does this using four For Next Loops. The first of these slides the screen on, inside this loop is another For Next Loop which is there to slow the first one down. If it was allowed to happen at its original speed, it would happen too quickly for your eyes to see the effect. The second set of For Next loops bounces the screen up and down a couple of times to add to the overall effect.

Procedure SCRN OFF[] lifts the screen off the viewing area ready for any large graphical change such as at the end of a level or end of a game. It allows you to update the display without the user seeing what is going on.

Now to make some pretty rainbows! The Procedure RAIN (it is NOT a typing error, it should be the number 1, not I.!))

We use rainbows to create an effective backdrop to the High score table. The whole procedure was created automatically by Rainbow Warrior, which is a program created by Martyn Brown. Martyn has given this program over to the Public Domain, so he does not get paid for it. If you use this, it will save you hours of work, so please be sure to credit him whenever you use it!

This delightful utility makes creating this type of special effect a dream. It allows you to draw the rainbows on screen in whatever form you want and then it saves out the whole procedure as an ascii file for you to use in your program.

It not only saves out the data in a form suitable for AMOS users, but also in other formats. These are Devpack, Binary, Decimal 1, Decimal 2, Hexadecimal 1, Hexadecimal 2, Kseka, Buffer and Screen formats, which must surely cover just about every development system.

First of all the Rainbow is set. This defines a rainbow affecting one specific colour, the length of the rainbow and its RGB value. Next the Rainbow command is used to create the rainbow by giving it a number, a base colour, the Y coordinate of where the effect is to start and the height of the rainbow.

In this program we are forcing the rainbow to affect only colour 0, we also want the borders around the viewing screen to be affected so that the whole screen is multicoloured, ie no black border. This is done by changing the colour of the border to 0 using the Colour Back command, so that now the

whole screen is coloured with rainbows. Now all that is needed is for the data to be loaded into the Rainbow command to produce the effect.

Now that we have made the pretty background, we move onto Procedure HI which displays the High score table.

The table of scores is stored out under sequential file format which makes it easy to load in and save out. To do this we open a channel to the disk drive using the Open In command, the number 1 defines the channel to be used, this is followed by the name of the file to be examined.

```
Open In 1, "hi-scores.s"
```

The '.s' just identifies the file as a sequential file to anyone looking at the disk directory.

We have already decided that there will be a maximum of 14 entries displayed on the Table, so we create a For Next loop of 14 to load in the 14 names and 14 scores which are stored in the variables HI\$() and HIN\$(). Once the For Next Loop is over, shut the channel that you have opened.

Procedure SRT is used to sort out the high scores into the right order. ie the best score is at the top, the lowest at the bottom.

Rather than sorting the variable that holds the scores and then trying to keep track of the name that goes with it, we were a bit sneaky here and created a third variable SRT\$(). The variables for the high score number and the name variable are stored in one string variable, this variable is then sorted with AMOS's inbuilt Sort command - SRT\$(0)- note here that you must use (0) and not any other number. This sorts the string giving the list from the lowest to the highest score, so next we must split the string and reverse the order. This is done with a For Next loop from 1 to 14 for the number of elements in the string. We are also working with a variable that works backwards so we create the variable TG=14.

Slicing strings is very simple, because we know that the numbers of the high scores are made up of six digits, we can split the string with

```
HIN$(Z) = Mid$(SRT$(TG), 1, 6)
```


This works in the following way. The variable HIN\$ (the first time this will be 1) is overwritten with the information in the middle of SRT\$(TG). The first time this will be the 14th element in the string, which will be the highest number.

Now that we know the number, it is easy to pull in the name.
`HI$(Z)=Upper$(Mid$(SRT$(TG),7,Len(SRT$(TG))))).`

This looks very complicated, but hopefully it can be explained easily.

HI\$(Z) the first time round Z=1. Next we tidy the users input by changing all the text to upper case with the Upper\$ command. Remember that TG will equal 14 on the first pass.

What happens is that it gets the middle of SRT\$(14) starting from the seventh character and going to the end of SRT(14). The result of that is turned into upper case and stored in HI\$(1). Then after all this is done, we Add TG,-1, which takes us backwards to TG(13) and we go onto the next Z.

The next procedure, WARN[2 parameters, both strings] creates the alert boxes which will warn the user about a possible mistake he or she is about to make. The routine was written by Aaron Fothergill of Shadow Software, and is a general purpose routine which can be taken out and used in your own programs, as long as you give credit to Aaron. It's been modified slightly to include indenting buttons and a shadowed effect on the text.

First of all we turn the mouse pointer with Show On. We now use two dimmed arrays which are restricted to this procedure - M\$(16) and B\$(4). M\$ is used for the lines of text, and B\$ is used for the buttons. Even though the routine is set up for 16 lines of text and 4 buttons, it is not recommended to use more than 8-10 lines of text and 2 buttons.

The work that this procedure does is quite complicated because it has been programmed to configure itself to the amount of lines and buttons as set up by the parameters, so that it knows where to place each line and button.

This procedure is completely self contained, and so we think that trying to explain it in detail would really be too lengthy and complicated. (Another

case of - If it works, use it!)

The buttons have been made to indent and the text shadowed using the same techniques used elsewhere in the program, eg on the options screen.

This is the end of the program listing, we hope that you have been able to follow through the explanations, and that we have given you some ideas to use in your own programs.

CHAPTER NINE

THE AMOS COMMANDS IN MARVIN



Set Buffer - Amos stores all your variables in a memory space of 8k, unless you tell it otherwise. The only limit to the size of arrays and string variables is the amount of memory your machine has. If the automatic setting is not big enough, you will get an error message saying 'out of string space'. Try increasing the amount of buffer space by 5k until the message goes away. If you get an 'out of memory' error while doing this, then you will have to look for ways of saving memory inside your program. **See Manual page 53.**

Screen Open - This command opens a screen and reserves memory for it. Therefore the more screens you open at any one time, the more memory is eaten up. When using this command, you must state the screen number, width (in pixels), height (in Pixels), number of colours, and mode (eg Lowres). The screen that has been opened will now be used for the following graphic and text operations in your program. **See Manual page 119.**

Curs Off - This turns off the cursor. **See Manual page 95.**

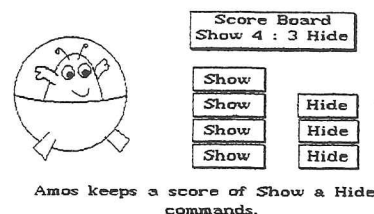
Flash Off - This turns the default flash colour off. In Amos this is set to colour 3. If it is not turned off before you load a picture, the colour numbered 3 in your palette will flash. **See Manual page 138.**

Cls - This will clear the screen to the colour number stated after it. eg Cls 0 clears the screen to colour 0. **See Manual page 131.**

Dim - This instruction is used at the start of a program to reserve space in memory for the variables used. It stands for Dimension and tells the computer that an array exists. **See Manual page 36.**

Global - This is used to define a list of variables which you need to use from within different procedures. This is very useful when you are writing a large program as the variables listed will not need a separate Shared statement. **See Manual page 46.**

Show - This brings back the mouse pointer after a Hide command. Amos remembers the amount of 'Show' and 'Hide' commands and if there are more hide than show commands, the pointer will remain hidden. If you want the pointer to appear straight away, use Show On. **See Manual page 165.**

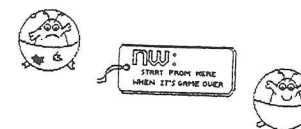


Variable - A variable is the place where the computer stores information for use in other parts of the program. A Numeric variable can only remember numbers, whereas a string variable can hold both numbers and letters and its name is followed by \$ (dollar sign). **See Manual page 35/36.**

Procedure - A procedure is a self contained set of instructions which you can call as a 'mini-program' from anywhere inside your main program. The procedure definition must start with Procedure followed by its name, it is ended by End Proc. Both these statements should be in a line of their own. For example, you could write a procedure that prints 'GAME OVER' on the screen, and then call for it to be used from any part of the program where this is required. **See Manual page 42/43.**

Label - As Amos does not use line numbers, labels are used to mark specific points in the listing. You can then return to the point marked by the label when necessary during the program. eg If your player dies and you want to

restart the game from a specific part of the program, you can mark the place with a name followed by a colon, and the program will go there if the event occurs. **See Manual page 41.**



If ... End if - This is known as a structured test. You can put a list of Amos instructions inside these terms and they will be repeated until a given condition is fulfilled. **See Manual page 76.**

Sam Play - This allows you to play a sound sample from an Amos sound bank. **See Manual page 235/6.**

While ... Wend - The While command allow you to repeat a series of instructions until a certain condition has been met. It can be used, for example, to move a sprite around a game continuously until it bumps into something, when the program sends it to a different instruction. The set of instructions used after While must be followed by Wend so that the program knows which actions to repeat. **See Manual page 78.**

Goto - This simply means Go To, whether to a line number, or a label. **See Manual page 73.**

MOD - MOD is a maths function that returns the remainder of a division calculation. **See Manual page 38.**

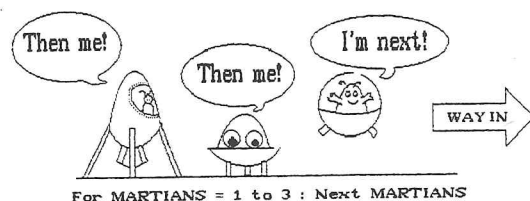
Map Plot - This is a TOME command

Icons - These are images which have been designed to produce background screens. They are permanently fixed on the screen and therefore you will not be able to move them around. **See Manual page 207.**

Paste Icon - This is the command used to draw an Icon onto the screen. The icon must have been stored in the icon bank (bank 2) and can be put

anywhere on the screen. **See Manual page 207.**

For ... Next - This is what is used to repeat a list of instructions a given number of times. For example, if you want to print your name 10 times on the screen without typing it in ten times, you would use a For Next loop, as follows:- For Z=1 to 10 You tell Z that it is counting from 1 -10 A\$="Marvin." Your name is stored as a string variable Print A\$ Print the information in A\$ Next Z Once completed, go on to the next value of Z. **See Manual page 77.**



Bob Clear - This removes all the bobs from the current screen and redraws the background they covered. **See Manual page 161.**

Screen Copy - This command is used to copy parts of a screen from one place to another very quickly. **See Manual page 132.**

Physical Screen/Logical Screen - These are terms that are only used when double buffer is in operation. The Physical screen (physic) is the one which is actually on view, the Logical (logic) is the screen where any changes can take place out of sight. **See manual 134/5.***

Screen Swap - This command switches the physical with the logical screen. ***See manual page 134.**

Double Buffer - If you use double buffer, then the above screen swapping will be done for you automatically. The Double Buffer command creates a second invisible copy of the screen being viewed. It uses this screen to perform the graphic operations such as bob movements without disturbing the picture being viewed. Moving bobs on a screen that is not double buffered results in flickering graphics. Remember, though that you will be

doubling the memory needed for each screen which is double buffered. **See Manual page 134/5.**

Bob - Bobs, like sprites can be moved around the screen without destroying the existing graphics. The only limit to size, number of colours and amount of bobs on a screen is the computer's memory. Bobs are slower than sprites, but they are more flexible. **See Manual page 155.**

Bob Draw - This will redraw a bob on the screen doing many things automatically for you. **See Manual page 161.**

Wait Vbl - This command halts the Amiga in its tracks and means wait for a vertical blank. It is used after certain commands such as Screen Swap so that the operation can be completed before the next instruction is started. **See Manual page 136.**

Bob Update - Bobs are automatically updated every 50th of a second, but sometimes you will need finer control. This is achieved by using Bob Update Off. This switches off the automatic updating and also any automatic screen swapping if it is being used. **See Manual page 161.**

Get Palette - This will get the palette from a chosen screen and saves it to the current screen. Using this, you can make sure that the source and destination screens are set to the same colours when moving information from one place to another. **See manual page 131.**

Get Block - Grabs a rectangular area of the screen which is stored as a block for later use. Each block is given a number so that it can be recalled as needed. **See Manual page 209.**

Put Block - This copies a block, identified by the number given when it was grabbed, to the current screen at a position specified in the x,y co-ordinates. If you leave out these co-ordinates, the block will be redrawn at its original position. **See Manual page 210.**

=Mouse Key - This checks to see if a mouse button has been pressed. **See Manual page 166.**

=Fire(j) - This will tell you if the fire button has been pressed on joystick

number(j) **See Manual page 166.**

Del Bob - This command is not in the manual! It allows you to delete bobs from your bob bank.

Erase - This will erase any bank (1-15) from memory, not from the disk. The memory space it occupied is then freed for use by the rest of the program. **See Manual page 50.**

Unpack - This decompresses a screen which has been compacted using the **Pack** or **Spack** command. Each of these has its own version of Unpack. **See Manual page 277.**

Screen Hide - This will temporarily hide a named screen from view, it can be brought back by using Screen Show. If the number of the screen to be hidden is not stated, the instruction is carried out on the current screen. **See Manual page 130.**

Open Out - This command will open a sequential file ready for writing, such as for the saving of high scores. If the file already exists it will be erased. **See Manual page 269.**

Screen Display - This will position a screen that you have defined using Screen Open anywhere on the viewing screen. This is the command that is used to create many of the bouncing screen effects seen in games and demos. **See Manual page 125.**

Appear - If you want to create a special effect while switching between two pictures, then this is the command to use. You can produce many interesting effects if you are willing to experiment with this command. **See Manual page 136.**

Channel - This assigns an animation (AMAL) channel to a screen object such as a bob. **See Manual page 197.**

Amal - AMos Animation Language. AMAL is used for smooth animation sequences which Basic cannot cope with. It is best used if the bobs have to repeatedly follow a predefined path such as the enemies in Marvin the Martian. The AMAL program is run independently and carries on by itself

while Amos goes on to do other things. **See Manual page 176.**

Move - This is an AMAL command that is used to move an object. **See Manual page 177.**

Reserve Zone - This will reserve memory for a given number of detection zones eg Reserve Zone 3. If the number is left out, then the current detection zones will be erased and the memory they used will be given back to the program. It should always be used before SET ZONE. **See Manual page 172.**

Set Zone - This will define a rectangular area of the screen which can be made to react to certain events in the program, eg a part of the screen that you click on to exit a game. **See Manual page 172.**

=Mouse Zone - This tells the computer the number of the zone that the mouse pointer has entered. **See Manual page 173.**

Paste Bob - This draws a copy of a specified image at a given point on the screen. **See Manual page 163.**

Text - Text displays graphical text on the screen starting from the given coordinates. Amos uses Topaz as its default font, but this can be changed by using the Set Font command. **See Manual page 106.**

Repeat ... Until - This will carry out a set of instructions until a given condition has been met. **See Manual page 78.**

Gosub - Means jump to a subroutine. Subroutines can be used in place of the more updated method of Procedures. They are a way of splitting your program into more manageable sized sections. The Gosub command can be used to jump to a given line, an Amos label or an expression such as a procedure name. **See Manual page 74.**

Return - Takes the program out of a previously called Subroutine and back to the line after the Gosub command. **See Manual page 74.**

Colour Back - This changes the colour of the border which you can see around the edges of the playing screen.

Paper - This chooses the background colour for your text. **See Manual page 87.**

Pen - This gives the colour for the text on your screen. **See Manual page 87.**

Ink - Sets the colours used by the drawing operations. **See Manual page 61.**

Rainbow Del - Switches off the Rainbow command. This is essential if you want rainbows in just one part of your program, if it is not turned off, it will affect the specified part of the screen throughout the program. This is not in the Amos manual.

Shared - This defines a list of variables inside a procedure. This means that they are then treated as Global and can be used by the main program. The variables must have been dimensioned at the start of your program. **See Manual page 45.**

Pop Proc - The usual way to exit a procedure is the End Proc command. If you should need to exit a procedure immediately, then Pop Proc is used. **See Manual page 47.**

Autoback - This is the automatic screen copying command. When using a double buffered screen the two screens must be synchronized to avoid nasty display effects such as flickering. There are three graphic modes that Autoback uses, and these are fully described in the manual. They are Autoback 2, which is the automatic mode and is default. Autoback 1 is the half automatic mode and Autoback 0 is manual, it stops the Autoback system working. All graphics are now drawn on the Logical screen. **See Manual page 158/9.**

=Rnd(n) - This is used to generate a random integer (whole number) between 0 and n. **See Manual page 115.**

Tile Val - This is another Tome command.

Reserve As Work - This reserves a temporary area of memory as working space. **See Manual page 49.**

Bload - Used for loading binary information into a specified bank or address. If Bloading into a bank, it must have already been defined and be large enough to hold the information. A bank that is too small will cause an overflow of information which can corrupt other areas of memory. **See Manual page 52.**

Timer - Timer is a reserved variable that increments by 1 every 50th of a second. It is often used to set things like the random generator. ****See Manual page 258.

Map Tile - A TOME command

View - This will display any updates to the current screen at the next vertical blank. This is used when Autoview is Off. **See Manual page 121.**

Rainbow - This creates the rainbow effects on the screen. The data for this can be created with Rainbow Warrior from Deja Vu Software. **See Manual page 141.**

Data - This precedes a list of useful information listed directly into a basic program. The information can then be loaded into variables using the Read command. **See Manual page 256.**

CHAPTER TEN

USING PROCEDURES IN YOUR OWN PROGRAMS

This version of Marvin the Martian contains the compiled game for you to play, the source code for you to look at and follow as you read this book and last, but not least, files containing individual procedures that you can merge into your own programs, as long as credit is given for each one in your programs as a REM line.

There are two ways in which you can put these into your own programs:-

By using the individual files on the disk, these are to be found in the directory 'Ripped_procs', they are saved as Ascii, so you will have to use 'merge as ascii' to put them into your programs. Do this by positioning the cursor where you want the procedure to be inserted in your program, go to the top options selector and click right mouse on Block Menu, then choose merge ascii. Go to the directory 'Ripped.procs' and load the procedure you want to merge, from the file selector, and it will appear in your program.

The exception to this is Procedure PSWD as this needs the pic.pac file on the disk. First merge the procedure PSWD.amos into your program, then enter Direct Mode and load pic.pac into an unused bank and save it along with your program.

Or you can rip them straight from the program listing as follows. This applies to all the procedures listed here, except PSWD. Using Procedure SDOWN [a,b,c] as an example. Go to the procedure name in the listing, if the procedure is shut, press F9 to open it. Put the cursor on the line with the procedure name, click right mouse and keeping it clicked, move down the listing until the line 'End Proc' is highlighted, then release the mouse button.

If you already have your program in Amos, go to block reserve then block store. Move to Edit other and choose your program from the selector, when it is on screen, position the cursor where you want to insert the procedure, go to block menu, then block paste to put into listing.

If you want to save the procedure to disk, highlight as before, then choose block menu and either block save or save as ascii. (Merge as ascii into your program as before if saved this way)

A procedure like PSWD has to be dealt with in a different way. This procedure needs a picture from a bank to make it work, so you must be loaded and saved out into a bank, making sure that the bank name corresponds with the one used in the procedure.

These are the procedures on the disk:-

PSWD - Procedure PSWD. The variable used is PWORD\$. You will need to alter the variable in the If End If condition 'If NBZ = 11 : End If' Alter all the conditions here to suit your own needs. The coding for the buttons will not need to be altered. The procedure also calls Procedures SCNOFF[], BNUMBERS and SCNON[]. These can either be added to your program or deleted from this procedure. BNUMBERS is used to compare the user's input with the password numbers stored in memory.

SDOWN - Procedure SDOWN[SPEED,SOURCE,DEST] This one is very easy to use, Delete the line beginning, If Source = 6 ... As long as you have two screens to work on, it can now be used.

WIPE - Procedure WIPE, this will work as it is, all you need is one screen with a picture on it.

RANDOM[] This will need a dimmed array at the beginning of your program, RD(number to randomise'). This will create a series of numbers, this will be the length of the parameters given earlier, there will be no repetition of numbers and no zeros.

SCNON and SCNOFF - Procedures SCNON[] and SCNOFF[] - These will both work as they are as long as there is a screen to view.

WARN - WARN[2 parameters, both strings] This one is also easy to rip out. Simply call it using the two strings. The first string contains the message, the split in the lines to be printed is defined with the '!' character (find it on the key next to the backspace arrow, use shift) This forces a new line to be printed. The second string is for the text printed on the buttons, the '!' is used

in the same way to separate the text.

We hope that you will find these procedures useful, and you will be able to use them in your own programs.

CHAPTER ELEVEN

AMOS HINTS AND TIPS

The following are hints and tips which we have gathered together from various sources over the past few months. They have come from various sources, we apologise for not crediting them to the various people who gave them to us, but it's been difficult to keep a track of who found what, so to everyone, wherever you are, Thank you!

Some of the tips may be familiar, some may be new, but we hope you will find something that will be of use to you.

Freezing your Mouse.

There are times that you will need to 'freeze' the Mouse pointer to make sure that the user cannot click while certain transactions are going on.

To freeze the pointer roughly in the middle of the screen,

Limit Mouse 272,162 to 272,162

Any coordinates work as long as the start and end coordinates are the same. Please note that these are hardware coordinates not screen coordinates. This will add 128 to the X axis and 50 to the Y axis.

Opening Workbench.

This tip is for the more advanced Amos users. If you want to open workbench from Amos, just use the following:-

A=Intcall(-210)

To close, A=Intcall(-78)

If you close Workbench using this method while Workbench has some windows open, when you reopen Workbench, the windows will be in the same state as you left them.

Not For The Faint-Hearted!

The following routines should only be used when you have truly come to grips with the Intcall and Execall functions as any errors will need a three-key reset to get out of the dreaded red box.

```
Dreg(0) = Execall(-132)
```

```
Dreg(0) = Execall(-138)
```

These, respectively turn multi-tasking on and off. They do have certain limitations, but the benefits greatly outweigh the drawbacks according to the type of game you are writing.

The main drawback is that when you turn off multi-tasking using this Execall, you lose ALL access to the keyboard. According to the version of Amos you are using, the mouse will also be locked until your program is compiled when it will work fine, compiling does not, however, bring back the keyboard. There is no effect on the joystick which will continue to function as before.

As an example, if you are creating a Shoot'em Up game where the keyboard is required to access certain functions eg changing weapons etc, then this cannot be used. If, however, your game has animated sequences with NO keyboard interaction, then this routine is for you.

You may have noticed that when using AMAL, under certain circumstances you get little shudders in your animations which are very difficult to eradicate. In the two weeks that we have been using this routine, the shuddering seems to have disappeared, there also seems to have been an increase of about 10% in the speed of the programs where it has been used.

So, all you have to do is to turn the keyboard off when multi-tasking isn't needed, and then back on when it is. We strongly recommend that this routine is not attached until the development of your program is over, (and then make a backup of the original which doesn't contain the routine before trying it!). Failure to get this right will only result in disaster should a run-time error occur while multi-tasking is turned off!

Fade Away.

If you are using the Fade or Palette command and all the parameters are the same number, eg for a 16 colour screen, the palette is \$FFF,\$FFF,\$FFF, (plus 13 more times) use

```
F = $FFF
```

Now all you have to do is type in Palette F,F,F (16 times in total). This will save wear and tear on your fingers and your keyboard!

Toggle It!

To toggle the value of a variable between 0 and 1 automatically, use this very handy little line:-

```
T = 1 - T
```

The value of T will be the opposite to that on the previous pass of the loop.

For Users of CTEXT.

This routine is for Ctext Users, but it could be converted to work with normal Text commands. It centres text on the horizontal axis.

```
PRNT: NBR = Plen(M$) Ctext160-(NBR/2),YT,M$ Return
```

To call this routine, use

```
YT = 100 M$ = "Hello AMOSers!" Gosub PRNT
```

Hang on a Minute!

This procedure makes the computer wait for a specified amount of time (contained in the variable NBR) or until the user hits a key.

```
Proc Wt[NBR] Repeat Add CV,1 K$=Inkey$ If K$ <> " ": CV=NBR: End  
If Wait 1 Until CV=NBR End Proc
```

This could also be used to check on the mouse key or the joystick.

eg If Mouse Key <>0: CV=NBR:End If

You would do similar if you were checking for the joystick.

Checking for a File.

If you are loading things from disk, it is always a good idea to put the file name inside a string and then check to see if the file exists in the following manner.

```
FIL$="Save.game" If exist(Fil$) Load FIL$ Else REM:Here you would
put an error trapping routine to tell REM:the user if the file
does not exist. End If
```

Hot Off The Presses!

At the time of writing, March 1992, a new Upgrade for Amos has just been released. This is Amos V1.34, and it alters the way in which some of the commands work.

According to the upgrade notices, VAL now works when compiled, this is something that would have been used in Marvin the Martian, but as it wouldn't compile, we had to find a way around it.

New commands Track Load and Track Play allow you to load and play a Sound Tracker or Noisetracker module without converting it to AMOS format first.

The Copy command will now work with odd addresses and odd lengths, which is very important and will save a lot of checking in your coding in the future.

You can play sound samples of any length.

There is an undocumented command called A = Prgstate. The value returned in A will be 0 if you are inside Amos, 1 if running under Ramos and -1 if the program has been compiled, this can be a handy thing to know in a program.

There is now a command called Multiwait, which makes the most of the Amiga's multitasking capabilities. Also included is Amos to Back which will push Amos behind any Workbench or CLI screen opened, Amos to Front does the opposite. Amos Here will return True if Amos is in view, and False if Workbench is in view.

Amos Lock and Amos Unlock disable or enable the Left Amiga + 'A' key system respectively. (These allow you to toggle between Amos and Workbench)

Bank Swap 1,2. This will swap two banks.

A=Display height. This command will return a value of 311 if in PAL mode, or 263 if in NTSC mode.

A=NTSC. This returns True if NTSC is being used, and False for PAL.

Request On, Request Off, Request WB. These, in order will turn the system requester on and off or will allow you to use the Workbench system requester.

Finally there is a full set of commands for using the Serial Device.

CHAPTER TWELVE

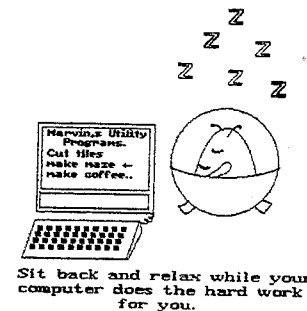
AMOS UTILITIES

In this chapter we would like to tell you a bit about the various programs we used or would have used if they had been available at the time, which make programming easier to cope with. After all, why should you spend hours and hours sweating over a problem when there is a utility program available to do it for you? With the exception of TOME itself, which is a commercial release, all the programs and extensions described here are either Public Domain, or Licensed Software, and are worthwhile additions to your programming 'toolkit'.

TOME has been featured throughout this book as it was the only thing that turned Marvin the Martian from an idea into a realistic prospect for a game. It should be a definite consideration for your 'toolkit' if you wish to get the best from AMOS.

You will find many shortcuts for yourselves as you progress with your knowledge of programming, you will soon say:-

"Why should I do that manually when I could write an Amos program/routine to do it for me?"

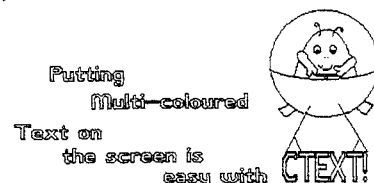


This is especially true of repetitive and time consuming tasks such as the collection of data for zones etc. All you have to do is to breakdown what you are doing and see if you can write your own program to do it for you.

Once you begin to think in this way, your coding speed will pick up and you will finish products a lot quicker than you would have thought possible. Many people must have given up programming because of the boring bits!

CTEXT.

An Amos extension by Aaron Fothergill, Shadow Software. Available as LPD56 from Deja Vu Software, price £3.50



Ctext is an extension for Amos written by Aaron Fothergill of Shadow Software. It allows you to use multi-colour fonts on the screen instead of disk based fonts.

Ctext stands for Colour Text and has been written to get rid of the problems caused by using different fonts in AMOS. It is installed onto your Amos Disk as extension number 8 and gives you 2 new commands and reserved variables. A new version of Ctext is being written at the same time as this book and will be available soon. This version will allow more than one font in a bank as well as allowing you to change the colour of the font itself, which will add even more variety to this flexible extension. It takes up less than 3000 bytes of memory and is one of the most powerful extensions so far.

The Font Size x,y command lets you set the size of your font. By using X or Y values of 0, it lets you make use of proportional fonts. Proportional fonts have differing widths and baselines for each character, in other words, an 'i' will be narrower than a 'w' etc.

The second command is Ctext x,y,string. Ctext uses this command in the same way as Amos uses its Text command, but it uses an icon bank for its font. This makes it easier to load in than a font and it does not fragment memory as disk fonts do.

It is used in addition to the text command, which is itself fast when using the

default font, but longwinded when a new font needs to be loaded from disk. Also, whereas Text command gives you single colour fonts only, CTEXT lets you use fonts of up to 64 colours. (Does that make your mind boggle?) It can also use any size font you want.

Ctext had not been released when we wrote Marvin the Martian, we only wish that it had! We would have used it to create the printing needed on the control panels and other displays such as the high score table.

As regards the control panel, we had to print everything twice to get an effect that resembles a two colour font. Remember how we described printing first in black, then overprinting one pixel offset on the X and Y positions to get a shadowed effect? We could have made really stylish displays with multi-coloured printing with very little effort using just one command! The display of text on the high score table would also have been much improved by using Ctext.

There are two support programs supplied with Ctext, the first is called font fixer, and the second is font scrunger.

The font scrunger program is used to cut out the characters from an IFF screen. The graphics for the font should be placed in horizontal rows on the screen with a gap between each letter and row. For Example:-

A B C D E F G H I J K

L M N O P Q R S T U V etc

Stored in this way, the font Scrunger will cut them all out for you. Once this has been done, you can use Amos' Sprite editor or Sprite X to remove the excess spaces. If you own Sprite X, then you can cut out the icons inside it as the scrunger has been built in to the program. This simple task saves a formidable amount of time.

The Font fixer program is much more involved, it allows you to assign a specific icon to a specific character. E.g. If your Icon number 1 happens to be the graphic for the letter 'a' (lower case), then Font Fixer allows you to inform the Ctext command that this is the case. You do this for the whole character set, not forgetting the spacebar, and now you have a whole new set of characters to use!

If you prefer to use proportionally spaced fonts, you can enter this information at the same time. The fonts are stored in Bank 10 as tiles and are very easy to manipulate as are all icons.

If you think, like I originally did, that Ctext was just for numbers and letters, think again! It is far more flexible than that!

Another use came to us while we were putting together Issue 2 of our disk magazine Totally Amos. We had just had the disply graphics redesigned for us by a professional graphic artist, Dicon Peeke of Wildfire Graphics, and we were attaching them to the magazine program. He had done such a good job, that we wanted to improve the general look of the magazine by giving it stylish looking indenting buttons.

While considering different methods of attaching these buttons to the magazine program, we realised that Ctext just grabs an icon and attaches it to a character, the icon, in theory, can be anything!

The magazine uses numbered buttons and a return key to select the different articles, a button for loading a file, one for going to the previous menu and two arrows which are used to scroll up and down the text.

So armed with this realisation, we cut out the buttons as icons 0-9, and a graphic each for the return key, the up and down arrows, the file selector and main menu button. We then loaded in the Fnt Fixer program and assigned these graphics to certain keys. The number keys were obviously assigned to the relevant number buttons, 'R' to the Return button, 'M' to the menu button and 'F' to the file selector button.

In the magazine program, once the zones were created around the graphic display, when checking for the mouse contacting a zone, eg user clicks on main menu button, we inserted the letter 'M' into a string and printed the icon with the Ctext command:-

```
Ctext 250,180, button$
```

but placed one pixel down and one pixel to the right of the original position. The program was stopped with -

```
Mouse Key <>0: Wend
```

then the buttons were printed back at their original position by printing Button\$ with Ctext. This was a very easy method of creating indenting buttons.

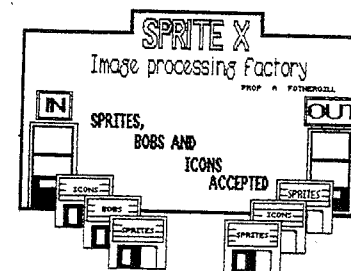
This method of using Ctext could be applied to different ideas with a bit of thought, and could make some areas of programming a lot easier.

Ctext also supports Kerning which is the name given to the process of moving individual characters nearer or farther away from each other. This means that you can control your text output completely.

SPRITE X

A Licenseware program by Aaron Fothergill, Shadow Software. Available as LPD55, from Deja Vu Software. Price £3.50

Sprite X is a much improved version of the sprite editor which is supplied with the Amos package.



As far as we can see this utility will do all you will ever need to do to sprites, bobs and icons for you.

Here is a list of the functions available to you, they all apply to bobs sprites and icons.

Load and Save.

Append i.e. add to the end of a bank,

Cut, It cuts in two different ways, the first way you can cut out bobs one at a time by dragging out a box around the bob. The second way automatically cuts a strip of bobs in the same way as the scrunger program supplied with Ctext as we described earlier.

Swap between the banks. ie swap icon bank to a sprite bank and a sprite bank to an icon bank.

Animate. This allows you to animate your bobs to test them before saving them so you can see if any alterations have to be made. This has been greatly improved since the early versions of this program. you can even load in an Iff picture to use as a backdrop to your animation.

Cut And Paste parts of the bob you are working on.

There are buttons which allow you to place the hot spot on your bob for collision detection purposes etc

The drawing facilities offered are very comprehensive offering dotted line, solid line, outline box, outline oval, line, spraycan, solid box, paint and text input.

You can flip a bob on the X or Y axis and move it in all four directions inside its box.

A bob can also be rotated 90° or to any number of degrees you wish.

You can clear a bob, set the resolution used from lowres 4 colour to lowres 64 colour all the way up to hires 4 colour to hires 16 colour.

Another essential feature is the undo button for those inevitable mistakes and a Niceness button which basically allows you to change the system foreground and background colours (ie the ones that Sprite X uses).

There is also the obligatory Credits button so you will know who created this essential utility!

You have a choice of noises to accompany every key click, a tone or bell can be used and the volume adjusted to suit your needs with a slider bar. A slider also controls the airbrush.

Here you will also find out how much chip mem you have left. There is no reference to Fast mem as all graphic work uses Chip memory, so it is irrelevant here. Fast mem is only used for the storage of packed screens etc.

There is also an Auto squash button which defaults to 'off'. This button automatically pushes the bob to the top left of its box so that as little memory as possible is wasted. The keyboard equivalent of this button is 'Z' which basically works its way through the whole bank of icons/bobs pushing them into the top left of their boxes.

Next we come to more buttons, one allows you to view a bob as a sprite, another allows you to widen the bottom box by multiples of 16 pixels on the X axis (this is the Amiga's limitations, not Sprite X's) and by a single pixel at a time on the Y axis. Two other buttons push the sprite to the top left of its box and push the box to its minimum limits

The + and - commands allow you to change the fill patterns for solid box etc. RGB allows you to change the RGB values of each colour. INS will insert a bob into a bank, and PUT places the bob you're working on into the bank. It is important to use this last facility as you will otherwise lose any changes you have made to that sprite.

GET BOB displays the bob of the current number in the editor box. The Dustbin button deletes a bob and the button next to it will delete a whole bank.

The next set of buttons are arrows which will take you around your sprite bank. If you press on the number between these arrows, the display changes to give you three boxes displaying the current bob plus two others.

The last button is QUIT which will not need explaining!

There are keyboard commands, among them 'C' will allow you to change the current colour selected to the background colour. 'B' makes a shadow of the current sprite. 'L' and 'S' are Load and Save respectively.

Sprite X is a very comprehensive sprite/bob/icon editor with as many facilities as you'd find in three or four other packages put together.

RAINBOW WARRIOR. A Public Domain program by Martyn Brown. Available from Deja Vu Software as APD 76.

Rainbow Warrior was created specifically to make using the Rainbow commands childishly easy. Creating these rainbow bars manually would be very time consuming, but Rainbow warrior does it all for you, in a way that's so easy that even a six year old would get good results. It even saves out the data in the form of listings that you can then name as a procedure and merge into your program so that it can be called when needed.

Once loaded in and run, Rainbow Warrior gives you all the instructions on the screen, and is controlled via the keyboard.

The idea is to draw the rainbows on the screen where you want them to appear and in the colours you choose.

The commands include Save, Load, Load IFF, increase and decrease the RGB values, clear the screen, store the current colour in memory, recall a colour from memory, next shade, darken and lighten a colour, random colours, popular colours, mirror, cut colours, paste buffer, fill region, refresh the screen, RAM save, screen, expand colours, and grab colours.

It is very easy to make a rainbow effect, to get you started we'll talk you through the steps needed to create a rainbow bar. You'll soon see how easy it is and soon you will be able to achieve stunning effects for your programs.

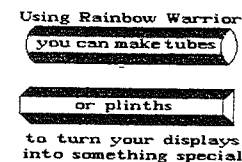
First of all, load and run Rainbow Warrior. Press the mouse button until the control panel appears, this has all the commands displayed on it.

On the bottom right of this panel you will see a box labelled COLOUR, this is the current colour, this can be changed in two ways. Pressing '/' will change the colour in the box through a range of popular colours, while pressing SHIFT+ '8' will bring up a random colour each time you press them.

Do this until you have a colour that you like. Now position the mouse pointer in the middle of the screen and press left mouse once. You will now get a line of your selected colour displayed right across the screen. Press the 'N' key which darkens the chosen colour and moves the cross-hairs down a line on

the screen. Press the mouse button again, but make sure that the mouse does not move. Repeat these two steps until the colour in the box goes to black, this usually takes 8 or 9 moves. Once you have done this you will see what looks like half of a tube, to create the second half, you can use a time saving feature which has been built into Rainbow Warrior. Place the mouse pointer on the first line that you created, and press the 'M' key. Now you should have some text displayed on the screen that tells you to select a region to be mirrored. Starting at the top line, hold down the mouse button and drag out a box going down to the last line. Once you release the button, you will be asked to select top, bottom or exit. Here we want to choose 'T' for top. This mirrors the rainbow to create the second half of your tube, it's magic!

To create a plinth effect, ie a tube with a flattened centre section where you can, for example, print text, is just as easy. Go to the middle of your tube and pressing right mouse move upwards. This will delete the top part of your tube and leave you with half a tube again.



Position the mouse pointer on the top line of the tube and press the 'H' key, this grabs the colour under the cross-hairs and gives you your original colour back. Now with the left button pressed, slide upwards about an inch making a solid band of colour. If you have moved too fast leaving gaps of black, simply go to the gap and press mouse button to fill it in.

Now put the mouse pointer on the very top line and press SHIFT + N then press left mouse, SHIFT + N and left mouse again. Remember to make sure that the mouse does not move until the colour in the box turns black again. This type of rainbow bar is very useful for displaying text.

It is important to remember that when you use rainbows, only one colour of your palette is affected. As an example, use DPaint create a 16 colour screen and paint it blue. Next place solid shapes on it using the background colour,

these can be boxes, circles or even text. Save out this screen.

Now use Rainbow Warrior to create a screen full of rainbows as described earlier and save out in Amos format. Next go into Amos and merge ascii, load the file saved from Rainbow Warrior. Once you have done this, you will see a listing, if you want this as a procedure, put the line Procedure RA1N at the beginning of the listing and a line End Proc at the bottom.

Now to test the results of your work, go to the top of the editor, press return to make a new line, move up to this line and key in the following

Load IFF "name of your test screen", 0 Proc RA1N Do Loop

Run the program and assuming that all has been done correctly you will see your IFF screen, where you drew shapes in your background colour you should now have your rainbows which you made in Rainbow Warrior. With careful planning you should be able to create stunning effects.

We have used Rainbow Warrior several times in our programs, and we have been really pleased with the results that it has created. You could use it to create metallic text on a title screen for example. You only have to remember to use the colour that the rainbow affects to display the rainbows. This defaults to colour 0, but it can be changed. Look for the Set Rainbow command in the listing and change the second parameter to the number of the colour you want. If you do not want the rainbows to affect the screen borders, do not use the colour 0. Rainbows only affects one colour, so to affect more colours you have to create more rainbows. This is a way of using far more than 16 colours on a 16 colour screen.

Rainbow Warrior is of use to coders who use other languages as well as Amos. It is also capable of saving out under Devpac, Kseka and C formats as well!

For any serious Amos user this program is a must (as well as for the other formats listed). Even if you just like messing about with screen effects, you will find that this program is invaluable as it makes life so much easier.

You can make rainbows move inside Amos's own animation language, AMAL. So you could easily create the multi-coloured Vu meter bars you see

bobbing up and down to the music in demos.

Very few games writers seem to use rainbows, they are seen as very pretty effects for demos and not much else. With a little imagination and planning, you could use Rainbow Warrior and its output to create really professional displays which will give your programs the polish needed to make them more eye-catching to your users.

TOME.

An AMOS extension by Aaron Fothergill, Shadow Software. Available directly from Shadow Software. Price £24.99

There are three programs supplied with the TOME package,

1. TOME itself.
2. Tile Maker.
3. Tile Paster.

Tome itself has been described throughout this book, the other two programs are very useful and deserve a mention as well.

The Tile Maker program is the program that cuts out the Tiles/Icons from an IFF screen and stores them in a bank ready for TOME to use.

This is an exceptionally easy program to use and is a very useful feature of the TOME package.

First of all you are given four buttons to select the size of the tiles you want to use. These are 16016, 32016, 16032 and 32032.

The next button is called Tile Tidy and we'll describe this in a moment.

A 'GO' button sets the tile-cutting machine into action,

The 'LOAD' button will load in tiles and give the facility of appending tiles to an existing bank.

You can also save tiles or load a picture.

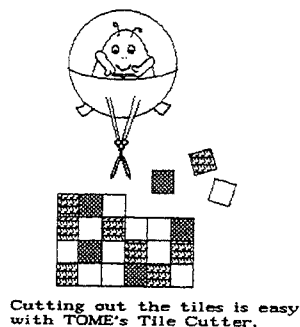
The 'Clear Tiles' option will erase the current tile bank.

'View picture' removes the control panel so you can see the whole screen,

Quit takes you out of the program.

Now back to the Tile Tidy feature which is the best thing about this program, it really saves the user a lot of hard work! This is what it does. When 'Tidy' is switched on, and the tiles have been cut out of the IFF picture, it searches through the whole tile bank for tiles that are blank. If it finds any it discards them. Its secondary task is to check each tile to see if any are identical, again if there are, the duplicates are taken out. If these tiles were left in the bank they would take up unnecessary memory.

In Marvin The Martian, this little function discarded seven tiles that were identical, so memory was saved. You will find out as your programming knowledge increases that memory conservation is vital for various reasons. Think of it as being as important as conservation of the environment is to our planet!



The next program is Tile Paster. This is a very simple and automatic program. It's only purpose is to load an Icon/tile bank, paste the tiles back on the screen and save them out as an IFF picture.

Both of these programs may sound unnecessary, especially the tile cutter, but apart from writing your own routine to cut out the tiles and discard the

duplicates, which would be a formidable task, the only other way we can see of doing this is to load them into Sprite X and grab the icons in that. That really would be long-winded.

As with all Aaron Fothergill's programs, the utilities that accompany them go to show that he doesn't only write great programs, he also tries to make the user's life as easy as possible!

AMOS 3D

The second Amos extension to come from Europress Software. Available from most outlets and directly from Europress. Price: around £34.99

With the Amiga games market overflowing with 3D games and selling very well, Europress decided it would be a good idea to allow all AMOS users the chance to write 3D games.

Amos 3D adds 30 new commands to AMOS. These new commands allow you to pull in your 3D world and move it, animate it and lots of other interesting things.

You also get OM or an Object Modelling program with the package. This is the main program and it allows you to create all the 3D graphics for use in your programs. It seems quite complex to use at first but once mastered it's like a dream.

Objects are built up just as though you were using a Lego set.

Amos 3D certainly deserves to do well and everyone who wants to write 3D games should buy this now!

AMOS COMPILER

The first commercial extension for AMOS from Europress. Available from most outlets and directly from Europress. Price: around £29.99

The main problem with AMOS was that it was too slow and anything written with it just couldn't stand up to the speed of commercial machine code games, that was, until Europress released this Compiler.

The main purpose for a compiler is to translate the program into machine code and save it to disc, and I am glad to say this is probably the best compiler ever produced.

It really is amazing to see your programs running before and after they have been compiled with this. Some games actually run too fast and you have to put in a routine to slow them down!!

The point is, if you own AMOS and use it on a regular basis, then don't consider this, BUY IT!!

Other programs to look out for are firstly, EASY AMOS, this is basically the same as AMOS but with a lot of commands taken out. Only buy this if you seriously cannot learn AMOS as some of the commands taken away are essential for games writing.

AMOS PROFESSIONAL is also due from Europress, this will be a professional development system for all serious AMOS users and looks likely to be at least £100 or more. It should be worth getting if you are developing games on a commercial basis.

APPENDIX ONE

ABOUT THE AUTHORS

We have been using AMOS since it was first released in June 1990. Since September 1990 we have released several programs, one is Public Domain, nine have been accepted onto Deja Vu Licensed Software and one has been released as a commercial product.

We have also worked with Europress Software on some of their products.

PAINT ME A STORY is an educational program in which children can create pictures either with the inbuilt graphics or the programs own art package. Text can then be added to the pictures, to make the pages of a book. There is a choice of magic wipes and fades to take you from one page to another and several pieces of music to go with the story.

The program is published by GENISOFT, Unit 3, Poyle 14, Newlands Drive, Colnbrook, BERKS, it cost £24.95.

PICK UP A PUZZLE (APD 103, Deja Vu Library) and its various data disks is a Public Domain Jigsaw game which has traditionally shaped jigsaw pieces.

The following are all Licensed Software, available from Deja Vu and authorised distributors.

THINGAMAJIG (LPD 4) This is a children's jigsaw puzzle game with large irregularly shaped pieces. there are three levels of difficulty and 24 puzzles to choose from.

JUNGLE BUNGLE (LPD 5) This is an icon driven adventure in which you have to find your way out of the jungle after a plane accident. It is aimed at children, but we know that several adults have had fun playing the game!

WORK & PLAY is a set of three educational programs. The first teaches the times tables by way of a board game, the second takes the child shopping where he/she has to answer simple arithmetic problems. The last teaches the

time both with an analogue clock and a digital watch, the aim is to get the time right so that the mouse can eat his fill of cheese without waking the cat.

JIGMANIA (LPD 13) is the Licenseware version of Pick up a Puzzle. It gives a choice of 100 or 35 piece puzzles to cater for adults and children. The extra feature is a construction kit which allows you to create jigsaws from your favourite IFF pictures. Several data disks are available on PD.

PLAY IT SAFE (LPD 14) In this program we try to make children, and their parents aware of dangers which could hurt them around the house, in the garden, in the park, on the street and in the car. You have to rid the house and other places of nasty beasties so that the family can move back in. There are five dangers in each place, and the child must help the teddy find them, each one found puts a part of the beast in a cage, when all five pieces are captured, the room is safe.

FLOWER POWER (LPD 27) This is another game created using TOME from Shadow Software. It is intended as 'Just for Fun', as it was such fun to create. The main character is a gardener who has entered the local flower show. You must help him dig the ground, sow the seed, water the growing plant, pick it, then take the required number of blooms to the flower show. At the same time you have to scare off all the bugs that want to eat the plants. There are 10 levels to complete to become the champion gardener.

BIG TOP FUN (LPD 29) Here we go to the circus to learn spelling and word recognition. There are 4 acts to see, Juggler has a unicyclist who needs help to balance the word he is holding with a picture from the screen. The next game is a version of the traditional pairs game, but here you have to match a word with a picture, two pictures or two words to make the pairs. Sealagrams is an anagram game. The seals balance lettered balls on their noses, you have to click on two seals to exchange the letters and make the word. In Balloon Burst the clown has to shoot the balloons and hit the letters making up a word.

SPARX'S STOCKING FILLERS (LPD 48) This is another 'Just For Fun disk'. It was released for Christmas 1991, but we hope that it will provide fun all year round. There are three programs here, the first is a seasonal version of Thingamajig, with Christmassy pictures to construct, there is also a small paint package so that you can also colour in the pictures. Another pairs program is included, you have to match up the designs on the Christmas

cards to make the pairs. The third program is another TOME created game. It is a sideways scroller where you have to help Santa deliver the presents down the chimneys. There are little beasties to hinder you and you will not be able to deliver a present to a house where the lights are still on.

MARVIN THE MARTIAN (LPD 49) We don't think that this one needs any further description! This version contains only the compiled game.

The future? Who knows!



There have been many people who have helped us since we started programming with AMOS, and we would like to send our special thanks to them.

FRANCOIS LIONET - For AMOS, he has created a whole new world and changed our lives.

EUROPRESS SOFTWARE - for publishing and continually supporting AMOS and also for helping us along the way and giving permission to use some of the sprites from Sprites 600.

AARON FOTHERGILL - for his help when we hit problem areas in our programs. We must also thank him for TOME which made Marvin the Martian a reality.

SANDRA SHARKEY - for supporting us, and ALL AMOS programmers. Sandra, through her Licensed Software and encouragement to do more, has given us the confidence to try doing things that we'd never even dreamed of before.

PETER HICKMAN - For his help in sorting out the Compiler v Marvin problem. Also for setting a target for us to aim at and giving help and encouragement.

PAUL TOWNSEND, of Technical Fred Software, - for his help and advise with AMOS problems.

MELANIE - for the initial idea for Marvin the Martian, and BEN and PIPPA for having the patience to test the program.

DAVID GIBBON of Software Developments - for giving us the opportunity to try and help other AMOS programmers who maybe have not got the access to so many people who could help them.

APPENDIX TWO

A LIST OF EVERY KNOWN AMOS CLUB - WORLDWIDE

The following list is not complete, but are all known to Europress Software. Many thanks to Nicola Murray at Europress for her hard work in gathering the list.

AUSTRALIA BELGIUM GERMANY

1. Amos User Club, 98 Carnarvan Street, Silverwater, New South Wales, 2141 Australia. TEL: 010 6127484700 FAX: 010 6127484684
2. Amos Belgium, Postbus 94, 9900 Eekto, Belgium.
3. DAUG, Deutsche AMOS Gruppe, Clubzentrale, Carsten Bernhard, Asternweg 4, 6229 Walluf, Germany.

ITALY NETHERLANDS SWEEDEN

1. Alessandro Gualtiari, Video Game, Via G Di Vittoria 1, ä20017 Mazzo Dhio Milan, Italy.
2. Amos Club, Kerkeind 8A, 5293 AB Gemonde, Netherlands.
3. Mac Larsson, PO BOX 6688, 5/11384, Stockholm, Sweeden.

UK

1. Len & Anne Tucker, Official AMOS Library, 1 Penmynydd Road, Penlan, Swansea. SA5 7EH TEL/FAX: 0792 588156
2. Deja Vu Software, 7 Hollingbrook, Beach Hill, Wigan. WN6 7SG TEL: 0942 495261

3. Aaron Fothergill, 1 Lower Moor, Whiddon Valley, Barnstaple, North Devon. EX32 8NW TEL: 0271 23544
4. All About AMOS, 36 Cleverly Estate, Wormholt Road, London. W12 OLX.
5. Paul Ciupek, NBS, 1 Chain Lane, Newport, Isle Of Wight. PO30 5QA TEL: 0983 529594

USA

1. Amos NTSC Club, 201/19 Tonnele Avenue, Gursey City. NJ 07306, USA.
2. David Lazarek, Computer People, 516 East 11th St, Michigan City, Indiana, IN 46360, USA. TEL: 0101 2198746380
3. Amos Club, PO BOX 11434, Micwavkae, WI 53211, USA.
4. Mary Hoffman, Safe Harbour, 2120 Moreland BVD, Suite L, Waukesha, WI 53186, USA. TEL: 0101 4145488120 FAX: 0101 4145488130
5. Amos Club, 185 RT206 BLDG 23/6, Flanders, MG 07836, USA.

To receive information from any of the above clubs, just send an s.a.e.

If you are looking to make some money from your programming efforts using AMOS, then contact NBS in the UK. They are offering to sell games/programs though their library, and you get paid!

APPENDIX THREE

HOW TO GET YOUR FREE GAME

To claim your free copy of Marvin The Martian, just send in the enclosed voucher which entitles you to one free copy of the game including the source code.

LOADING INSTRUCTIONS

To load in Marvin The Martian and play it just put the the drive and it will automatically load.

SETTING UP YOUR AMOS DISK TO ACCEPT THE SOURCE CODE.

All files with Marvin. in the directory can be loaded into AMOS.

If you already own TOME, then you can go ahead and load the source code for Marvin the Martian.

TOME is a commercial product, not Public Domain, so to allow us to give you the source code, Aaron Fothergill has given us an adapted version of the Tome lib file for you to add to your AMOS system disk. **THIS DOES NOT MEAN THAT YOU HAVE A VERSION OF TOME!** It will allow you to load in the source code of any other Tome products, but as the information supplied is specific to Marvin, loading anything else into this version of TOME could cause the Amiga to Guru.

If you **DO NOT** own TOME, you will have to follow the steps below.

You must be working with Amos V1.3 or later. If you have an older Amos, then this will not work. Updates are always available from Deja Vu Software.

MAKE A BACKUP of your Amos Disk, do not work on the original.

Load your back up copy of AMOS, and load the Config program which

allows you to customise AMOS to your own needs. Run the program then using the right mouse on the top menu bar, select 'Disc' and load the default configuration. When the message 'Choose menu option' comes back, select 'Set' on top menu bar and go to loaded extensions.

Select number 7 from the list displayed and key in the following (exactly as it is here)

: AMOS_System/Tome_L.Lib

Once this is done, select 'quit' and go to 'Disc' on the menu bar and save the configuration. Reset your Amiga and copy a file called Tome_L.Lib from the root directory of the Marvin disk, and put in the drawer AMOS_System. Once this is done, Amos is ready to accept Marvin the Martian.

Doing this will not affect the way in which AMOS works if the file is left on the disk, but if you purchase TOME and attach it at a later date, copy Tome's lib file from the Tome disk to your Amos disk. At the time of writing this is called Tome.Lib. Put it in the Amos system directory of Amos, then delete Tome_L.Lib, load in the config program into Amos as before, select Loaded Extensions and alter Tome_L.Lib to Tome.Lib. To remove the altered Tome from your Amos, delete the file Tome_L.Lib and delete the line entered in the loaded extensions option in the config program.

If you have any problems then please write to me, Len Tucker, at the address given in Chapter One, enclose an s.a.e. I will try and sort out any problems you have loading the source code or game.

amos

TOME



SHADOW
SOFTWARE

AMOS TOME is just one of those things you have to have if you're serious AMOS" Phil South. Amiga Shopper March 1992

You've read how AMOS TOME was used to write Marvin the Martian, now you have the full blown version and use it for your own games, (ALL the major programmers are using TOME, including Aaron Fothergill, Len Tucker, Peter Hickman, Symons, Stephen Hill and more). It is compatible with the Compiler and AMOS upwards. TOME now has 40 commands, and the editor is still the most powerful editor available on ANY computer.

TOME has already been used to write 7 commercial games (at the last count), many more are being developed. 90% of games use map based backgrounds,

g AMOS TOME the most useful extension for AMOS.

The TOME Goodies disk 1 is available to registered TOME users, with a 3D action game, 3D Isometric scrolling game and the source code to Magic Forest horizontally scrolling platform game (recently included in Amiga Format's list of essential 10 P.D games). All 3 are written using TOME.

The Goodies Disk 1 normally costs £5.00, but you can get it free when you order AMOS TOME with the order form below.

AMOS TOME Series IV is the latest version and costs £29.99 (AMOS Club members get a discount) and is only available from Shadow Software in the U.K. Overseas customers, please contact us for details of more local distributors).

When ordering AMOS TOME, please make cheques payable to Shadow Software. Send your order to: Shadow Software, 1 Lower Moor, Whiddon Valley, Barnstaple, Devon. EX32 8NW. England.

All U.K. prices are inclusive of V.A.T at 17.5% and postage & packing.

AMOS TOME requires AMOS V1.3 or above, and the TOME Editor requires 1 Megabyte of memory.

I send me AMOS TOME (V4.0) with Free TOME Goodies disk !!

Name

Address

.....

.....

Postcode

Close a Cheque/Postal Order for £29.99. If ordering from overseas, please make sure that payment is in British Pounds and drawn on a London Bank. (Eurocheques and U.S Postal Money orders made out in pounds are also accepted).

DEJA Vu Software

7 Hollingbrook, Beach Hill, Wigan, WN6
0942 495261 STRICTLY MAIL ORDER

PROGRAMS LISTED ARE
COMPATIBLE WITH
THE AMIGA A500 PLUS

Now in our third year of trading

{DEJA VU Professional Licensed Software Titles}

(* denotes 1Mb)

{PD Titles}

TITLES FOR THE KIDS

LPD4 Thingmajig
LPD5 Jungle Bungle *
LPD8 Work & Play *
LPD10 Word factory
LPD15 Play It Safe *
LPD27 Flower Power *
LPD29 Big Top Fun *
LPD37 Rocket Maths *
LPD45 Music Box *
LPD48 Stocking Fillers *
LPD49 Marvin the Martian *
LPD51 Magical Young Artist
LPD59 Prehistoric Fun Pack *
LPD70 Paintbox *
LPD72 Monster Island *
LPD82 Col. Book II *
LPD83 Picture Hangman *

MISCELLANEOUS

LPD42 X-Stitch *
LPD52 LC24/200 Fonts
LPD73 AMOS Database
LPD75 Video Lab *
LPD81 Pools Pro V1.1

GAMES

LPD11 Go-Getter *
LPD17 Dogfight II *
LPD19 X-It-50
LPD23 E.S.P.
LPD47 Dirty Cash *
LPD66 Hotel Manager *
LPD67 Cyadonia *
LPD69 Magical Mix-Up
LPD71 Battlecars *
LPD74 Sour Grapes *
LPD76 T-Tecmaze *
LPD84 Guess Who *
LPD85 Magic Wassoaks *
LPD87 Puzword *

FOR AMOS USERS

LPD40 Sprite Bank Editor
LPD55 SpriteX *
LPD56 CText *
LPD79 Music Engine *
LPD80 Icon Bank Editor
LPD86 NCOMMAND

NOW AVAILABLE

FRED FISH 400 - 600
AMOS PD 1 -371

SPECIAL PD PACKS

Pack 1 AMOS Program discs 1
- 10

Pack 2 AMOS Program Discs
11 - 20

Pack 3 AMOS Program discs
21 - 30

Special price to AMOS In
Action Owners £15.00 per pack

AMOS PD DISKS

APD36 AMOS Updater disc
APD76 Rainbow Warrior
APD200 Dungeon Master Shell
APD214 Hanissis V Demo 1 *
APD255 Hanissis V Demo 2 *
APD371 AMOS 3D programs

LICENSED SOFTWARE PRICES:

£3.50 each UK (incl VAT), £3.75 each Europe, £4.00 each ROW

All prices include a royalty payment to the programmer.

PD PRICES:

£1.75 per disc UK (incl VAT), £2 Europe, £2.25 ROW

Choose 1 PD disc free with every 10 discs

POSTAGE & PACKING:

Add 50p P&P to total cost of order

Cheques/PO's should be crossed and made payable to Deja Vu Software

Send £1 for our catalogue disc for details of our Clip Art Collection, General PD and
Official AMOS PD list

AMOS In Action

This excellent new book gives ideas, tips and inside information on writing good games with AMOS, the world renowned games writing package for the Amiga. Details, write-ups and opinions on the add-on packages for AMOS are also included - which are the best and which do you most need? Also included are contacts for shareware libraries and information on how to get your games published.

Len Tucker is an Amiga fanatic who regularly contributes to shareware and licenceware libraries, he has gained such a reputation for himself through licenceware that he now works extensively for major commercial companies.

Invaluable information for all AMOS users from an accomplished and **successful** programmer.

All this and a free game (with source code) too! A coupon for a free disc is included as part of the book.

Published by
Kuma

Kuma Computers Ltd,
Pangbourne, Berkshire, England
Tel: 0734 - 844335
Fax: 0734 - 844339

£12.95 net



AMOS In Action
Anne & Len Tucker
KUMA

AMOS

In Action

Anne & Len Tucker